Visual C#

Lucian Sasu

May 30, 2005

Cuprins

1 Platforma Microsoft .NET **9**
- 1.1 Prezentare generală . . . 9
- 1.2 Arhitectura platformei Microsoft .NET . . . 11
- 1.3 Componente ale lui .NET Framework . . . 12
  - 1.3.1 Microsoft Intermediate Language . . . 12
  - 1.3.2 Common Language Specification . . . 12
  - 1.3.3 Common Language Runtime . . . 13
  - 1.3.4 Common Type System . . . 14
  - 1.3.5 Metadata . . . 15
  - 1.3.6 Assemblies . . . 16
  - 1.3.7 Assembly cache . . . 17
  - 1.3.8 Garbage collection . . . 17
- 1.4 Trăsături ale platformei .NET . . . 17

2 Tipuri predefinite, tablouri, string-uri **21**
- 2.1 Vedere generală asupra limbajului C# . . . 21
- 2.2 Tipuri de date . . . 23
  - 2.2.1 Tipuri predefinite . . . 24
  - 2.2.2 Tipuri valoare . . . 25
  - 2.2.3 Tipul enumerare . . . 29
- 2.3 Tablouri . . . 36
  - 2.3.1 Tablouri unidimensionale . . . 36
  - 2.3.2 Tablouri multidimensionale . . . 37
- 2.4 Şiruri de caractere . . . 41
  - 2.4.1 Expresii regulate . . . 43

3 Clase, instrucţiuni, spaţii de nume **45**
- 3.1 Clase – vedere generală . . . 45
- 3.2 Transmiterea de parametri . . . 49
- 3.3 Conversii . . . 55
  - 3.3.1 Conversii implicite . . . 55

3.3.2 Conversiile implicite ale expresiilor constante 57
3.3.3 Conversii explicite 57
3.3.4 Boxing şi unboxing 60
3.3.5 Declaraţii de variabile şi constante 62
3.3.6 Declaraţii de etichete 62
3.3.7 Instrucţiuni de selecţie 63
3.3.8 Instrucţiuni de ciclare 64
3.3.9 Instrucţiuni de salt 66
3.3.10 Instrucţiunile try, throw, catch, finally 66
3.3.11 Instrucţiunile checked şi unchecked 67
3.3.12 Instrucţiunea lock 67
3.3.13 Instrucţiunea using 67
3.4 Spaţii de nume 68
3.4.1 Declaraţii de spaţii de nume 69
3.4.2 Directiva using 70

4 Clase **75**
4.1 Declararea unei clase 75
4.2 Membrii unei clase 76
4.3 Câmpuri 77
4.3.1 Câmpuri instanţe 77
4.3.2 Câmpuri statice 77
4.3.3 Câmpuri readonly 78
4.3.4 Câmpuri volatile 78
4.3.5 Iniţializarea câmpurilor 79
4.4 Constante 79
4.5 Metode 80
4.5.1 Metode statice şi nestatice 81
4.5.2 Metode externe 81
4.6 Proprietăţi 81
4.7 Indexatori 87
4.8 Operatori 92
4.8.1 Operatori unari 93
4.8.2 Operatori binari 95
4.8.3 Operatori de conversie 95
4.8.4 Exemplu: clasa Fraction 96
4.9 Constructori de instanţă 99
4.10 Constructori statici 99
4.11 Clase interioare 101

5 Destructori. POO în C# 105
5.1 Destructori . . . 105
5.2 Specializarea şi generalizarea . . . 107
5.2.1 Specificarea moştenirii . . . 107
5.2.2 Apelul constructorilor din clasa de bază . . . 108
5.2.3 Operatorii *is* şi *as* . . . 110
5.3 Clase *sealed* . . . 110
5.4 Polimorfismul . . . 111
5.4.1 Polimorfismul parametric . . . 111
5.4.2 Polimorfismul ad–hoc . . . 111
5.4.3 Polimorfismul de moştenire . . . 111
5.4.4 *Virtual* şi *override* . . . 113
5.4.5 Modificatorul *new* pentru metode . . . 114
5.4.6 Metode *sealed* . . . 116
5.4.7 Exemplu folosind *virtual, new, override, sealed* . . . 117
5.5 Clase şi metode abstracte . . . 119
5.6 Interfeţe . . . 120
5.6.1 Clase abstracte sau interfeţe? . . . 126

6 Delegaţi. Evenimente. Structuri 127
6.1 Tipul delegat . . . 127
6.1.1 Utilizarea delegaţilor pentru a specifica metode la run-time . . . 128
6.1.2 Delegaţi statici . . . 131
6.1.3 Multicasting . . . 132
6.2 Evenimente . . . 135
6.2.1 Publicarea şi subscrierea . . . 135
6.2.2 Evenimente şi delegaţi . . . 135
6.2.3 Comentarii . . . 142
6.3 Structuri . . . 142
6.3.1 Structuri sau clase? . . . 146

7 Tratarea excepţiilor. Atribute 147
7.1 Tratarea excepţiilor . . . 147
7.1.1 Tipul *Exception* . . . 147
7.1.2 Aruncarea şi prinderea excepţiilor . . . 148
7.1.3 Reîncercarea codului . . . 159
7.1.4 Compararea tehnicilor de manipulare a erorilor . . . 161
7.1.5 Sugestie pentru lucrul cu excepţiile . . . 162
7.2 Atribute . . . 163
7.2.1 Generalităţi . . . 163

7.2.2 Atribute predefinite . . . 164
7.2.3 Exemplificarea altor atribute predefinite . . . 167
7.2.4 Atribute definite de utilizator . . . 170

8 ADO.NET 173
8.1 Ce reprezintă ADO.NET? . . . 173
8.2 Furnizori de date în ADO.NET . . . 174
8.3 Componentele unui furnizor de date . . . 174
8.3.1 Clasele *Connection* . . . 175
8.3.2 Clasele *Command* . . . 176
8.3.3 Clasele *DataReader* . . . 176
8.3.4 Clasele *DataAdapter* . . . 176
8.3.5 Clasa *DataSet* . . . 176
8.4 Obiecte *Connection* . . . 176
8.4.1 Proprietăţi . . . 177
8.4.2 Metode . . . 179
8.4.3 Evenimente . . . 179
8.4.4 Stocarea stringului de conexiune în fişier de configurare 179
8.4.5 Gruparea conexiunilor . . . 180
8.5 Obiecte *Command* . . . 181
8.5.1 Proprietăţi . . . 181
8.5.2 Metode . . . 182
8.5.3 Utilizarea unei comenzi cu o procedură stocată . . . 185
8.5.4 Folosirea comenzilor parametrizate . . . 185
8.6 Obiecte *DataReader* . . . 187
8.6.1 Proprietăţi . . . 187
8.6.2 Metode . . . 188
8.6.3 Crearea şi utilizarea unui *DataReader* . . . 188
8.6.4 Utilizarea de seturi de date multiple . . . 190
8.6.5 Accesarea datelor într–o manieră sigură din punct de vedere a tipului . . . 190

9 ADO.NET (2) 193
9.1 Obiecte *DataAdapter* . . . 193
9.1.1 Metode . . . 194
9.1.2 Proprietăţi . . . 195
9.2 Clasa *DataSet* . . . 195
9.2.1 Conţinut . . . 196
9.2.2 Clasa *DataTable* . . . 196
9.2.3 Relaţii între tabele . . . 199
9.2.4 Popularea unui *DataSet* . . . 199

9.2.5 Propagarea modificărilor către baza de date 200
9.3 Tranzacţii în ADO.NET 202

10 Fire de execuţie **205**
10.1 Managementul thread–urilor 205
10.1.1 Pornirea thread–urilor 205
10.1.2 Metoda *Join()* 208
10.1.3 Suspendarea firelor de execuţie 209
10.1.4 Omorârea thread–urilor 209
10.1.5 Sugerarea priorităţilor firelor de execuţie 213
10.1.6 Fire în fundal şi fire în prim-plan 213
10.2 Sincronizarea 215
10.2.1 Clasa *Interlocked* 217
10.2.2 Instrucţiunea *lock* 218
10.2.3 Clasa *Monitor* 219

11 ASP.NET **223**
11.1 Anatomia unei pagini ASP.NET 224
11.1.1 Adăugarea unui control Web 224
11.1.2 Adăugarea scriptului *inline* 225
11.1.3 Code-behind 226
11.2 Clasa *Page* 228
11.2.1 Evenimentul *Init* 228
11.2.2 Evenimentul *Load* 228
11.2.3 Evenimentul *Unload* 229
11.3 Crearea unei pagini ASP.NET folosind Visual Studio.NET . . 229
11.4 Controale server 231
11.4.1 Postback 231
11.4.2 Data Binding 233
11.5 Controale Web 233
11.5.1 Label 234
11.5.2 Button 234
11.5.3 LinkButton 235
11.5.4 TextBox 235
11.5.5 CheckBox 235
11.5.6 RadioButton 235
11.5.7 DropDownList 235

12 ASP.NET (2) **237**
12.1 Controale de validare 237
12.1.1 RequiredFieldValidator 238

12.1.2 RegularExpressionValidator . . . 238
12.1.3 RangeValidator . . . 238
12.1.4 CompareValidator . . . 238
12.1.5 CustomValidator . . . 239
12.1.6 ValidationSummary . . . 240
12.2 Comunicarea cu browserul . . . 240
12.2.1 Generarea programatică conţinutului . . . 241
12.2.2 Redirectarea . . . 241
12.3 Cookies . . . 242
12.4 Fişierul de configurare al aplicaţiei Web . . . 243
12.5 Fişierul *global.asax* . . . 243
12.6 Managementul sesiunii . . . 244
12.7 ViewState . . . 245
12.7.1 Application . . . 245
12.8 Alte puncte de interes în aplicaţii ASP.NET . . . 245

13 Servicii Web **247**
13.1 Generalităţi . . . 247
13.1.1 Componentele unui WS . . . 247
13.2 Discuţii asupra blocurilor componente . . . 248
13.2.1 Protocolul de transport . . . 248
13.2.2 Schema de codificare . . . 248
13.2.3 Convenţia de formatare . . . 249
13.2.4 Mecanismul de descriere . . . 250
13.2.5 Mecanisme de descoperire . . . 250
13.3 Crearea unui serviciu Web simplu . . . 251
13.3.1 Adunarea a două numere . . . 251
13.3.2 Validarea unui număr de card de credit . . . 252
13.3.3 SW pentru transmitere de conţinut binar . . . 253
13.4 SOAP . . . 254
13.4.1 Header-e . . . 255
13.4.2 Elementul *body* . . . 256

Curs 1

Platforma Microsoft .NET

1.1 Prezentare generală

Platforma .NET este un nou cadru de dezvoltare a softului, sub care se vor realiza, distribui si rula aplicaţiile de tip forme Windows, aplicaţii WEB si servicii WEB. Ea constă în trei părţi principale: Common Language Runtime, clase pentru platformă şi ASP.NET. O infrastructură ajutătoare, Microsoft .NET Compact Framework este un set de interfeţe de programare care permite dezvoltatorilor realizarea de aplicaţii pentru dispozitive mobile precum telefoane inteligente şi PDA-uri[1].

.NET Framework constituie un nivel de abstractizare între aplicaţie şi kernel-ul sistemului de operare (sau alte programe), pentru a asigura portabilitatea codului; de asemenea integrează tehnologii care au fost lansate de catre Microsoft incepând cu mijlocul anilor 90 (COM, DCOM, ActiveX, etc) sau tehnologii actuale (servicii Web, XML).

Platforma constă câteva grupe de produse:

1. *Unelte de dezvoltare* - un set de limbaje (C#, Visual Basic .NET, J#, Managed C++, Objective-C, Python, Smalltalk, Eiffel, Perl, Fortran, Cobol, Lisp, Haskell, Pascal, RPG, etc), un set de medii de dezvoltare (Visual Studio .NET, Visio), infrastructura .NET Framework, o bibliotecă cuprinzătoare de clase pentru crearea serviciilor Web (Web Services)[2], aplicaţiilor Web (Web Forms) şi aplicaţiilor Windows (Windows Forms).

2. *Servere specializate* - un set de servere Enterprise .NET, continuatoare ale lui SQL Server 2000, Exchange 2000, BizTalk 2000, etc, care pun la

[1]Definiţia oficială Microsoft, martie 2005, www.microsoft.com/net/basics/glossary.asp
[2]Serviciilor Web - aplicaţii care oferă servicii folosind Web-ul ca modalitate de acces.

dispoziţie funcţionalităţi diverse pentru stocarea bazelor de date, email, aplicaţii B2B[3].

3. *Servicii Web* - cel mai notabil exemplu este .NET Passport - un mod prin care utilizatorii se pot autentifica pe site-urile Web vizitate, folosind un singur nume şi o parolă pentru toate. Deşi nu este omiprezent, multe site-uri îl folosesc pentru a uşura accesul utilizatorilor.

4. *Dispozitive* - noi dispozitive non–PC, programabile prin .NET Compact Framework, o versiune redusă a lui .NET Framework: Pocket PC Phone Edition, Smartphone, Tablet PC, Smart Display, XBox, set-top boxes, etc.

Motivul pentru care Microsoft a trecut la dezvoltarea acestei platforme este maturizarea industriei software, accentuându–se următoarele direcţii:

1. *Aplicaţiile distribuite* - sunt din ce în ce mai numeroase aplicaţiile de tip client / server sau cele pe mai multe nivele ($n-$tier). Tehnologiile distribuite actuale cer de multe ori o mare afinitate faţă de producător şi prezintă o carenţă acută a interoperării cu Web-ul. Viziunea actuală se depărtează de cea de tip client/server către una în care calculatoare, dispozitive inteligente şi servicii conlucrează pentru atingerea scopurilor propuse. Toate acestea se fac deja folosind standarde Internet neproprietare (HTTP, XML, SOAP).

2. *Dezvoltarea orientată pe componente* - este de mult timp cerută simplificarea integrării componentelor software dezvoltate de diferiţi producători. COM (Component Object Model) a realizat acest deziderat, dar dezvoltarea şi distribuirea aplicaţiilor COM este prea complexă. Microsoft .NET pune la dispoziţie un mod mai simplu de a dezvolta şi a distribui componente.

3. *Modificări ale paradigmei Web* - de-a lungul timpului s–au adus îmbunătăţiri tehnologiilor Web pentru a simplifica dezvoltarea aplicaţiilor. În ultimii ani, dezvoltarea aplicaţiilor Web s–a mutat de la prezentare (HTML şi adiacente) către capacitate sporită de programare (XML şi SOAP).

4. *Alţi factori de maturizare a industriei software* - reprezintă conştientizarea cererilor de interoperabilitate, scalabilitate, disponibilitate; unul din deziderateleе .NET este de a oferi toate acestea.

[3]Bussiness to Bussiness - Comerţ electronic între parteneri de afaceri, diferită de comerţul electronic între client şi afacere (Bussiness to Consumer – B2C).

1.2 Arhitectura platformei Microsoft .NET

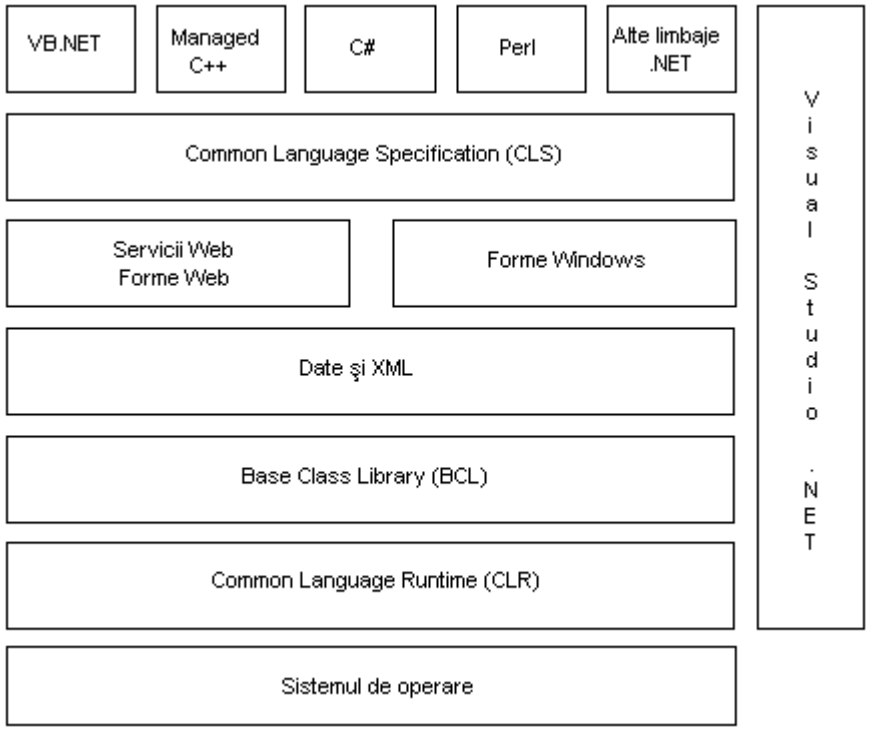

Figura 1.1: Arhitectura .NET

Figura 1.1 schematizează arhitectura platformei Microsoft .NET. Orice program scris într-unul din limbajele .NET este compilat în Microsoft Intermediate Language (MSIL), în concordanţă cu Common Language Specification (CLS). Aceste limbaje sunt sprijinite de o bogată colecţie de biblioteci de clase, ce pun la dispoziţie facilităţi pentru dezvoltarea de Web Forms, Windows Forms şi Web Services. Comunicarea dintre aplicaţii şi servicii se face pe baza unor clase de manipulare XML şi a datelor, ceea ce sprijină dezvoltarea aplicaţiilor cu arhitectură *n-tier*. Base Class Library există pentru a asigura funcţionalitate de nivel scăzut, precum operaţii de I/O, fire de execuţie, lucrul cu şiruri de caractere, comunicaţie prin reţea, etc. Aceste clase sunt reunite sub numele de .NET Framework Class Library, ce permite dezvoltarea rapidă a aplicaţiilor. La baza tuturor se află cea mai importantă componentă a lui .NET Framework - Common Language Runtime, care răspunde de execuţia fiecărui program. Mediul de dezvoltare Visual Studio .NET nu este absolut necesar pentru a dezvolta aplicaţii (există şi alternative open-source pentru el), dar datorită extensibilităţii este cel mai

bine adaptat pentru crearea aplicaţiilor. Evident, nivelul inferior este rezervat sistemului de operare. Trebuie spus că platforma .NET nu este exclusiv dezvoltată pentru sistemul de operare Microsoft Windows, ci şi pentru arome de Unix (FreeBSD sau Linux - a se vedea proiectul Mono[4]).

1.3 Componente ale lui .NET Framework

1.3.1 Microsoft Intermediate Language

Una din uneltele de care dispune ingineria software este *abstractizarea*. Deseori vrem să ferim utilizatorul de detalii, să punem la dispoziţia altora o interfaţă simplă, care să permită atingerea scopului, fără a fi necesare cunoaşterea tuturor detaliilor. Dacă interfaţa rămâne neschimbată, se pot modifica toate detaliile interne, fără a afecta acţiunile celorlaţi beneficiari ai codului.

În cazul limbajelor de programare, s-a ajuns treptat la crearea unor nivele de abstractizare a codului rezultat la compilare, precum *p-code* (cel produs de compilatorul Pascal-P) şi *bytecode* (binecunoscut celor care au lucrat în Java). Bytecode-ul Java, generat prin compilarea unui fişier sursă, este cod scris într-un limbaj intermediar care suportă POO. Bytecod-ul este în acelaşi timp o abstractizare care permite executarea codului Java, indiferent de platforma ţintă, atâta timp cât această platformă are implementată o maşină virtuală Java, capabilă să "traducă" mai departe fişierul class în cod nativ.

Microsoft a realizat şi el propria sa abstractizare de limbaj, aceasta numindu-se Common Intermediate Language. Deşi există mai multe limbaje de programare de nivel înalt (C#, Managed C++, Visual Basic .NET, etc), la compilare toate vor produce cod în acelaşi limbaj intermediar: Microsoft Intermediate Language (MSIL, sau IL pe scurt). Asemănător cu bytecod-ul, IL are trăsături OO, precum abstractizarea datelor, moştenirea, polimorfismul, sau concepte care s-au dovedit a fi extrem de necesare, precum excepţiile sau evenimentele. De remarcat că această abstractizare de limbaj permite rularea aplicaţiilor independent de platformă (cu aceeaşi condiţie ca la Java: să existe o maşină virtuală pentru acea platformă).

1.3.2 Common Language Specification

Unul din scopurile .NET este de a sprijini integrarea limbajelor astfel încât programele, deşi scrise în diferite limbaje, pot interopera, folosind din plin moştenirea, polimorfismul, încapsularea, excepţiile, etc. Dar limbajele nu

[4] www.go-mono.com

sunt identice: unele suportă supraîncărcarea operatorilor (Managed C++, C#), altele nu (Visual Basic .NET); unele sunt case sensitive, altele nu. Pentru a se asigua totuşi interoperabilitatea codului scris în diferite limbaje, Microsoft a publicat Common Language Specification (CLS), un subset al lui CTS (Common Type System, vezi 1.3.4), conţinând specificaţii de reguli necesare pentru integrarea limbajelor.

CLS defineşte un set de reguli pentru compilatoarele .NET, asigurând faptul că fiecare compilator va genera cod care interferează cu platforma (mai exact, cu CLR–ul — vezi mai jos) într-un mod independent de limbajul sursă. Obiectele şi tipurile create în diferite limbaje pot interacţiona fără probleme suplimentare. Combinaţia CTS/CLS realizează de fapt interoperarea limbajelor. Concret, se poate ca o clasă scrisă în C# să fie moştenită de o clasă scrisă în Visual Basic care aruncă excepţii ce sunt prinse de cod scris în C++ sau J#.

1.3.3 Common Language Runtime

CLR este de departe cea mai importantă parte componentă a lui .NET Framework. Este responsabil cu managementul şi execuţia codului scris în limbaje .NET, aflat în format IL; este foarte similar cu Java Virtual Machine. CLR instanţiază obiectele, face verificări de securitate, depune obiectele în memorie, disponibilizează memoria prin garbage collection.

În urma compilării unei aplicaţii rezultă un fişier cu extensia exe, dar care nu este un executabil portabil Windows, ci un executabil portabil .NET (.NET PE). Acest cod nu este deci un executabil nativ, ci se va rula de către CLR, întocmai cum un fişier class este rulat în cadrul JVM. CLR foloseşte tehnologia compilării JIT - o implementare de maşină virtuală, în care o metodă sau o funcţie, în momentul în care este apelată pentru prima oară, este tradusă în cod maşină. Codul translatat este depus într-un cache, evitând-se astfel recompilarea ulterioară. Există 3 tipuri de compilatoare JIT:

1. *Normal JIT* - a se vedea descrierea de mai sus.

2. *Pre-JIT* - compilează întregul cod în cod nativ singură dată. În mod normal este folosit la instalări.

3. *Econo-JIT* - se foloşeşte pe dispozitive cu resurse limitate. Compilează codul IL bit cu bit, eliberând resursele folosite de codul nativ ce este stocat în cache.

În esenţă, activitatea unui compilator JIT este destinată a îmbunătăţi performanţa execuţiei, ca alternativă la compilarea repetată a aceleiaşi bucăţi de cod în cazul unor apelări multiple. Unul din avantajele mecanismului JIT apare în clipa în care codul, o dată ce a fost compilat, se execută pe diverse procesoare; dacă maşina virtuală este bine adaptată la noua platformă, atunci acest cod va beneficia de toate optimizările posibile, fără a mai fi nevoie recompilarea lui (precum în C++, de exemplu).

1.3.4 Common Type System

Pentru a asigura interoperabilitatea limbajelor din .NET Framework, o clasă scrisă în C# trebuie să fie echivalentă cu una scrisă în VB.NET, o interfaţă scrisă în Managed C++ trebuie să fie perfect utilizabilă în Managed Cobol. Toate limbajele care fac parte din pleiada .NET trebuie să aibe un set comun de concepte pentru a putea fi integrate. Modul în care acest deziderat s-a transformat în realitate se numeşte Common Type System (CTS); orice limbaj trebuie să recunoască şi să poată manipula nişte tipuri comune.

O scurtă descriere a unor facilităţi comune (ce vor fi exhaustiv enumerate şi tratate în cadrul prezentării limbajului C#):

1. *Tipuri valoare* - în general, CLR-ul (care se ocupă de managementul şi execuţia codului IL, vezi mai jos) suportă două tipuri diferite: tipuri valoare şi tipuri referinţă. Tipurile valoare reprezintă tipuri alocate pe stivă şi nu pot avea valoare de null. Tipurile valoare includ tipurile primitive, structuri şi enumerări. Datorită faptului că de regulă au dimensiuni mici şi sunt alocate pe stivă, se manipuleaza eficient, reducând overhead-ul cerut de mecanismul de garbage collection.

2. *Tipuri referinţă* - se folosesc dacă variabilele de un anumit tip cer resurse de memorie semnificative. Variabilele de tip referinţă conţin adrese de memorie heap şi pot fi null. Transferul parametrilor se face rapid, dar referinţele induc un cost suplimentar datorită mecanismului de garbage collection.

3. *Boxing şi unboxing* - motivul pentru care există tipuri primitive este acelaşi ca şi în Java: performanţa. Însă orice variabilă în .NET este compatibilă cu clasa Object, rădăcina ierarhiei existente în .NET. De exemplu, *int* este un alias pentru *System.Int32*, care se derivează din *System.ValueType*. Tipurile valoare se stochează pe stivă, dar pot fi oricând convertite într-un tip referinţă memorat în heap; acest mecanism se numeşte *boxing*. De exemplu:

```
int i = 1; //i - un tip valoare
object box = i; //box - un obiect referinta
```

Când se face boxing, se obţine un obiect care poate fi gestionat la fel ca oricare altul, făcându–se abstracţie de originea lui.

Inversa boxing-ului este unboxing-ul, prin care se poate converti un obiect în tipul valoare echivalent, ca mai jos:

```
int j = (int)box;
```

unde operatorul de conversie este suficient pentru a converti de la un obiect la o variabilă de tip valoare.

4. *Clase, proprietăţi, indexatori* - platforma .NET suportă pe deplin programarea orientată pe obiecte, concepte legate de obiecte (încapsularea, moştenirea, polimorfismul) sau trăsături legate de clase (metode, câmpuri, membri statici, vizibilitate, accesibilitate, tipuri interioare, etc). De asemenea se includ trăsături precum proprietăţi, indexatori, evenimente.

5. *Interfeţe* - reprezintă acelaşi concept precum clasele abstracte din C++ (dar conţinând doar funcţii virtuale pure), sau interfeţele Java. O clasă care se derivează dintr-o interfaţă trebuie să implementeze toate metodele acelei interfeţe. Se permite implementarea simultană a mai multor interfeţe (în rest moştenirea claselor este simplă).

6. *Delegaţi* - inspiraţi de pointerii la funcţii din C, ce permit programarea generică. Reprezintă versiunea "sigură" a pointerilor către funcţii din C/C++ şi sunt mecanismul prin care se tratează evenimentele.

Există numeroase componente ale lui CLR care îl fac cea mai importantă parte a lui .NET Framework: metadata, assemblies, assembly cache, reflection, garbage collection.

1.3.5 Metadata

Metadata înseamnă în general "date despre date". În cazul .NET, ea reprezintă informaţii destinate a fi citite de către maşină, nu de utilizatorul uman. Este stocată împreună cu codul pe care îl descrie. Pe baza metadatei, CLR ştie cum să instanţieze obiectele, cum să le apeleze metodele, cum să acceseze proprietăţile. Printr-un mecanism numit reflection, o aplicaţie (nu

neapărat CLR) poate să interogheze această metadată şi să afle ce expune un obiect.

Mai pe larg, metadata conţine o declaraţie a fiecărui tip şi câte o declaraţie pentru fiecare metodă, câmp, proprietate, eveniment al tipului respectiv. Pentru fiecare metodă implementată, metadata conţine informaţie care permite încărcătorului clasei respective să localizeze corpul metodei. De asemena mai poate conţine declaraţii despre *cultura* aplicaţiei respective, adică despre localizarea ei (limba folosită în partea de interfaţă utilizator). Mecanismul prin care aceste informaţii se folosesc pe parcursul rulării de către aplicaţii se numeşte *reflection*.

1.3.6 Assemblies

Un assembly reprezintă un bloc funcţional al unei aplicaţii .NET. El formează *unitatea fundamentală de distribuire, versionare, reutilizare şi permisiuni de securitate*. La runtime, un tip de date există în interiorul unui assembly (şi nu poate exista în exteriorul acestuia). Un assembly conţine metadate care sunt folosite de către CLR. Scopul acestor "assemblies" este să se asigure dezvoltarea softului în mod "plug-and-play". Dar metadatele nu sunt suficiente pentru acest lucru; mai sunt necesare şi "manifestele".

Un manifest reprezintă metadate despre assembly-ul care găzduieşte tipurile de date. Conţine numele assembly-ului, informaţia despre versiune, referiri la alte assemblies, o listă a tipurilor în assembly, permisiuni de securitate şi altele.

Un assembly care este împărţit între mai multe aplicaţii are de asemenea un *shared name*. Această informaţie care este unică este opţională, neapărând în manifestul unui assembly daca acesta nu a fost gândit ca o aplicaţie partajată.

Deoarece un assembly conţine date care îl descriu, instalarea lui poate fi făcută copiind assemblyul în directorul destinaţie dorit. Când se doreşte rularea unei aplicaţii conţinute în assembly, maniferstul va instrui mediul .NET despre modulele care sunt conţinute în assembly. Sunt folosite de asemenea şi referinţele către orice assembly extern de care are nevoie aplicaţia.

Versionarea este un aspect deosebit de important pentru a se evita aşa–numitul "DLL Hell". Scenariile precedente erau de tipul: se instaleaza o aplicaţie care aduce nişte fişiere .dll necesare pentru functionare. Ulterior, o altă aplicaţie care se instalează suprascrie aceste fişiere (sau măcar unul din ele) cu o versiune mai nouă, dar cu care vechea aplicaţie nu mai funcţionează corespunzător. Reinstalarea vechii aplicaţii nu rezolvă problema, deoarece a doua aplicaţie nu va funcţiona. Deşi fişierele dll conţin informaţie relativ la versiune în interiorul lor, ea nu este folosită de către sistemul de operare,

ceea ce duce la probleme. O soluţie la această dilemă ar fi instalarea fişierelor dll în directorul aplicaţiei, dar în acest mod ar dispărea reutilizarea acestor biblioteci.

1.3.7 Assembly cache

Assembly cache este un director aflat în mod normal în directorul `\Winnt \Assemblies`. Atunci când un assembly este instalat pe o maşină, el va fi adăugat în assembly cache. Dacă un assembly este descărcat de pe Internet, el va fi stocat în acest assembly cache, într-o zonă tranzientă. Aplicaţiile instalate vor avea assemblies într-un assembly cache global.

În acest assembly cache vor exista versiuni multiple ale aceluiaşi assembly. Dacă programul de instalare este scris corect, va evita suprascrierea assembly-urilor deja existente (şi care funcţionează perfect cu acplicaţiile instalate), adăugând doar noul assembly. Este un mod de rezolvare a problemei "DLL Hell", unde suprascrierea unei biblioteci dinamice cu o variantă mai nouă putea duce la nefuncţionarea corespunzătoare a aplicaţiilor anterior instalate. CLR este cel care decide, pe baza informaţiilor din manifest, care este versiunea corectă de assembly de care o aplicaţie are nevoie. Acest mecanism pune capăt unei epoci de tristă amintire pentru programatori şi utilizatori.

1.3.8 Garbage collection

Managementul memoriei este una din sarcinile cele mai consumatoare de resurse umane. Garbage collection este mecanismul care se declanşează atunci când alocatorul de memorie răspunde negativ la o cerere de alocare de memorie. Implementarea este de tip "mark and sweep": se presupune iniţial că toată memoria alocată se poate disponibiliza, dupa care se determină care din obiecte sunt referite de variabilele aplicaţiei; cele care nu mai sunt referite sunt dealocate, celelalte zone de memorie sunt compactate. Obiectele a căror dimensiune de memorie este mai mare decât un anumit prag nu mai sunt mutate, pentru a nu creşte semnificativ penalizarea de performanţă.

În general, CLR este cel care se ocupă de apelarea mecanismului de garbage collection. Totuşi, la dorinţă, programatorul poate cere rularea lui.

1.4 Trăsături ale platformei .NET

Prin prisma celor prezentate pâna acum putem rezuma următoarele trăsături:

- **Dezvoltarea multilimbaj:** Deoarece există mai multe limbaje pentru această platformă, este mai uşor de implementat părţi specifice în limbajele cele mai adecvate. Numarul limbajelor curent implementate este mai mare decât 10. Această dezvoltare are în vedere şi debugging-ul aplicaţiilor dezvoltate în mai multe limbaje.

- **Independenţa de procesor şi de platformă:** IL este independent de procesor. O dată scrisă şi compilată, orice aplicaţie .NET (al cărei management este făcut de către CLR) poate fi rulată pe orice platformă. Datorită CLR-ului, aplicaţia este izolată de particularităţile hardware sau ale sistemului de operare.

- **Managementul automat al memoriei:** Problemele de tipul memory leakage nu mai trebuie să preocupe programatorii; overhead-ul indus de către mecanismul de garbage collection este suportabil, fiind implementat în sisteme mult mai timpurii.

- **Suportul pentru versionare:** Ca o lecţie învăţată din perioada de "DLL Hell", versionarea este acum un aspect de care se ţine cont. Dacă o aplicaţie a fost dezvoltată şi testată folosind anumite componente, instalarea unei componente de versiune mai nouă nu va atenta la buna funcţionare a aplicaţiei în discuţie: cele două versiuni vor coexista paşnic, alegerea lor fiind făcută pe baza manifestelor.

- **Sprijinirea standardelor deschise:** Nu toate dispozitivele rulează sisteme de operare Microsoft sau folosesc procesoare Intel. Din această cauză orice este strâns legat de acestea este evitat. Se foloseşte XML şi cel mai vizibil descendent al acestuia, SOAP. Deoarece SOAP este un protocol simplu, bazat pe text, foarte asemănător cu HTTP[5], el poate trece uşor de firewall-uri, spre deosebire de DCOM sau CORBA.

- **Distribuirea uşoară:** Actualmente instalarea unei aplicaţii sub Windows înseamnă copierea unor fişiere în nişte directoare anume, modificarea unor valori în regiştri, instalare de componente COM, etc. Dezinstalarea completă a unei aplicaţii este in majoritatea cazurilor o utopie. Aplicaţiile .NET, datorită metadatelor şi reflectării trec de aceste probleme. Se doreşte ca instalarea unei aplicaţii să nu însemne mai mult decât copierea fişierelor necesare într-un director, iar dezinstalarea aplicaţiei să se facă prin ştergerea acelui director.

- **Arhitectură distribuită:** Noua filosofie este de a asigura accesul la servicii Web distribuite; acestea conlucrează la obţinerea informaţiei

[5]Folosit la transmiterea paginilor Web pe Internet

dorite. Platforma .NET asigură suport şi unelte pentru realizarea acestui tip de aplicaţii.

- **Interoperabilitate cu codul "unmanaged":** Codul "unmanaged" se referă la cod care nu se află în totalitate sub controlul CLR. El este rulat de CLR, dar nu beneficiază de CTS sau garbage collection. Este vorba de apelurile funcţiilor din DLL-uri, folosirea componentelor COM, sau folosirea de către o componentă COM a unei componente .NET. Codul existent se poate folosi în continuare.

- **Securitate:** Aplicaţiile bazate pe componente distribuite cer automat securitate. Modalitatea actuală de securizare, bazată pe drepturile contului utilizatorului, sau cel din Java, în care codul suspectat este rulat într-un "sandbox", fără acces la resursele critice este înlocuit în .NET de un control mai fin, pe baza metadatelor din assembly (zona din care provine - ex. Internet, intranet, maşina locală, etc) precum şi a politicilor de securitate ce se pot seta.

Curs 2

Vedere generală asupra limbajului C#. Tipuri predefinite. Tablouri. Şiruri de caractere

2.1 Vedere generală asupra limbajului C#

C#[1] este un limbaj de programare modern, simplu, obiect–orientat. Este foarte asemănător cu Java şi C++, motiv pentru care curba de învăţare este foarte lină.

Un prim exemplu de program, care conţine o serie de elemente ce se vor întâlni în continuare, este:

```
using System;
class HelloWorld
{
  public static void Main()
  {
    Console.WriteLine(‘‘Hello world!’’);
  }
}
```

Prima linie *using System;* este o directivă care specifică faptul că se vor folosi clasele care sunt incluse în spaţiul de nume[2] *System*; un spaţiu de nume este o colecţie de tipuri sau o grupare de alte spaţii de nume care pot

[1] "#" se pronunţă "sharp"

[2] Eng: namespace

fi folosite într-un program (detalii vor fi date mai târziu). În cazul de faţă, clasa care este folosită din acest spaţiu de nume este *Console*. Mai departe, orice program este conţinut într–o clasă, în cazul nostru *HelloWorld*. Punctul de intrare în aplicaţie este metoda *Main*, care nu preia în acest exemplu nici un argument din linia de comandă şi nu returnează explicit un indicator de stare a terminării.

Pentru operaţiile de intrare–ieşire cu consola, se foloseşte clasa *Console* din spaţiul de nume *System*; pentru această clasă se apelează metoda statică *WriteLine*, care afişează pe ecran mesajul, după care face trecerea la linie nouă.

O variantă uşor modificată a programului de mai sus, în care se salută persoanele ale căror nume este transmis prin linia de comandă:

```
using System;
class HelloWorld
{
  public static void Main( String[] args)
  {
    for( int i=0; i<args.Length; i++)
    {
      Console.WriteLine( ‘‘Hello {0}’’, args[i]);
    }
  }
}
```

În exemplul de mai sus metoda principală preia o listă de parametri transmişi din linia de comandă (un şir de obiecte de tip *String*) şi va afişa pentru fiecare nume “Hello” urmat de numele de indice i (numerotarea parametrilor începe de la 0). Construcţia “{0}” va fi înlocuită cu primul argument care urmează după “Hello {0}”. La executarea programului de mai sus în forma: `HelloWorld Ana Dan`, se va afişa pe ecran:

```
Hello Ana
Hello Dan
```

Metoda Main poate să returneze o valoare întreagă, care să fie folosită de către sistemul de operare pentru a semnala dacă procesul s–a încheiat cu succes sau nu[3]. Menţionăm faptul că limbajul este case–sensitive[4].

Ca metode de notare Microsoft recomandă folosirea următoarelor două convenţii:

[3]De exemplu în fişiere de comenzi prin testarea variabilei de mediu ERRORLEVEL

[4]Face distincţie între litere mari şi mici

- Conventie Pascal, în care prima literă a fiecărui cuvânt se scrie ca literă mare
- convenţia tip "cămilă" este la fel ca precedenta, dar primul caracter al primului cuvânt nu este scris ca literă mare.

În general, convenţia tip Pascal este folosită pentru tot ce este vizibil din exteriorul unei clase, precum clase, metode, proprietăţi, etc. Doar parametrii metodelor se scriu cu convenţia cămilă. Se recomandă evitarea folosirii notaţiei ungare (numele unei entităţi trebuie să se refere la semantica ei, nu la tipul de reprezentare).

Cuvintele cheie (cuvinte rezervate care nu pot fi folosite ca şi identificatori) din C# sunt date în tabelul 2.1:

Tabelul 2.1: Cuvinte cheie în C#

abstract	base	bool	break	byte
case	catch	char	checked	class
const	continue	decimal	default	delegate
do	double	else	enum	event
explicit	extern	false	finally	fixed
float	for	foreach	goto	if
implicit	in	int	interface	internal
is	lock	long	namespace	new
null	object	operator	out	override
params	private	protected	public	readonly
ref	return	sbyte	sealed	short
sizeof	static	string	struct	switch
this	throw	true	try	typeof
uint	ulong	unchecked	unsafe	ushort
using	virtual	void	while	

2.2 Tipuri de date

C# prezintă două tipuri de date: *tipuri valoare* şi *tipuri referinţă*. Tipurile valoare includ tipurile simple (ex. *char, int, float*)[5], tipurile enumerare

[5]De fapt acestea sunt structuri, prezentate în alt curs

şi structură şi au ca principale caracteristici faptul că ele conţin direct datele referite şi sunt alocate pe stivă sau *inline* într–un struct. Tipurile referinţă includ tipurile clasă, interfaţă, delegat şi tablou, toate având proprietatea că variabilele de acest tip stochează referinţe către obiecte. Demn de remarcat este că toate tipurile de date sunt derivate (direct sau nu) din tipul System.Object, punând astfel la dispoziţie un mod unitar de tratare a lor.

2.2.1 Tipuri predefinite

C# conţine un set de tipuri predefinite, pentru care nu este necesară includerea vreunui spaţiu de nume via directiva `using`: string, object, tipurile întregi cu semn şi fără semn, tipuri numerice în virgulă mobilă, tipurile bool şi decimal. Tipul string este folosit pentru manipularea şirurilor de caractere codificate Unicode; conţinutul obiectelor de tip `string` nu se poate modifica[6]. Clasa object este rădăcina ierarhiei de clase din .NET, la care orice tip (inclusiv un tip valoare) poate fi convertit.

Tipul bool este folosit pentru a reprezenta valorile logice true şi false. Tipul char este folosit pentru a reprezenta caractere Unicode, reprezentate pe 16 biţi. Tipul decimal este folosit pentru calcule în care erorile determinate de reprezentarea în virgulă mobilă sunt inacceptabile, el punând la dispoziţie 28 de cifre zecimale semnificative.

Tabelul de mai jos conţine lista tipurilor predefinite, arătând totodată cum se scriu valorile corespunzătoare:

Tabelul 2.2: Tipuri predefinite.

Tip	Descriere	Exemplu
object	rădacina oricărui tip	object a = null;
string	o secvenţă de caractere Unicode	string s = “hello”;
sbyte	tip întreg cu semn, pe 8 biţi	sbyte val = 12;
short	tip întreg cu semn, pe 16 biţi	short val = 12;
int	tip întreg cu semn, pe 16 biţi	int val = 12;
long	tip întreg cu semn, pe 64 biţi	long val1 = 12; long val2=34L;
byte	tip întreg fără semn, pe 8 biţi	byte val = 12;
ushort	tip întreg fără semn, pe 16 biţi	ushort val = 12;
uint	tip întreg fără semn, pe 32 biţi	uint val = 12;

[6]Spunem despre un string că este *invariabil* - engl. *immutable*

Tabelul 2.2(continuare)

Tip	Descriere	Exemplu

ulong	tip întreg fără semn, pe 64 de biţi	ulong val1=12;
		ulong val2=34U;
		ulong val3=56L;
		ulong va;4=76UL;

float	tip cu virgulă mobilă, simplă precizie	float val=1.23F;
double	tip în virgulă mobilă, dublă precizie	double val1=1.23; double val2=4.56D;
bool	tip boolean	bool val1=false; bool val2=true;
char	tip caracter din setul Unicode	char val='h';
decimal	tip zecimal cu 28 de cifre semnificative	decimal val=1.23M;

Fiecare din tipurile predefinite este un alias pentru un tip pus la dispoziţie de sistem. De exemplu, string este alias pentru clasa System.String, int este alias pentru System.Int32.

2.2.2 Tipuri valoare

C# pune programatorului la dispoziţie tipuri valoare, care sunt fie structuri, fie enumerări. Există un set predefinit de structuri numite *tipuri simple*, identificate prin cuvinte rezervate. Un tip simplu este fie de tip numeric[7], fie boolean. Tipurile numerice sunt tipuri întregi, în virgulă mobilă sau decimal. Tipurile intregi sunt sbyte, byte, short, ushort, int, uint, long, ulong, char; cele în virgulă mobilă sunt float şi double. Tipurile enumerare se pot defini de către utilizator.

Toate tipurile valoare derivează din clasa System.ValueType, care la rândul ei este derivată din clasa object (alias pentru System.Object). Nu este posibil ca dintr–un tip valoare să se deriveze. O variabilă de tip valoare întotdeauna trebuie să conţină o valoare (nu poate fi null, spre deosebire de tipurile referinţă). Atribuirea pentru un astfel de tip înseamnă copierea valorii (clonarea) dintr–o parte în alta.

Structuri

Un tip structură este un tip valoare care poate să conţină declaraţii de constante, câmpuri, metode, proprietăţi, indexatori, operatori, constructori sau tipuri imbricate. Vor fi descrise în într–o secţiune următoare.

[7]Engl: integral type

Tipuri simple

C# pune are predefinit un set de tipuri structuri, numite tipuri simple. Tipurile simple sunt identificate prin cuvinte rezervate, dar acestea reprezintă doar alias–uri pentru tipurile struct corespunzătoare din spaţiul de nume System; corespondenţa este dată în tabelul de mai jos:

Tabelul 2.3: Tipuri simple şi corespondenţele lor cu tipurile din spaţiul de nume System.

Tabelul 2.3

Cuvânt rezervat	Tipul alias
sbyte	System.SByte
byte	System.Byte
short	System.Int16
ushort	System.UInt16
int	System.Int32
uint	System.UInt32
long	System.Int64
ulong	System.UInt64
char	System.Char
float	System.Single
double	System.Double
bool	System.Boolean
decimal	System.Decimal

Deoarece un tip simplu este un alias pentru un tip struct, orice tip simplu are membri. De exemplu, tipul int, fiind un tip alias pentru System.Int32, următoarele declaraţii sunt legale:

```
int i = int.MaxValue;    //constanta System.Int32.MaxValue
string s = i.ToString(); //metoda System.Int32.ToString()
string t = 3.ToString(); //idem
//double d = Double.ParseDouble("3.14");
```

Tipuri întregi

C# suportă nouă tipuri întregi: sbyte, byte, short, ushort, int, uint, long, ulong şi char. Acestea au următoarele dimensiuni şi domeniu de valori:

- sbyte reprezintă tip cu semn pe 8 biţi, cu valori de la -128 la 127;

- byte reprezintă tip fără semn pe 8 biţi, între 0 şi 255;
- short reprezintă tip cu semn pe 16 biţi, între -32768 şi 32767;
- ushort reprezintă tip fără semn pe 16 biţi, între 0 şi 65535;
- int reprezintă tip cu semn pe 32 de biţi, între -2^{31} şi $2^{31} - 1$;
- uint reprezintă tip fără semn pe 32 de biţi, între 0 şi $2^{32} - 1$;
- long reprezintă tip cu semn pe 64 de biţi, între -2^{63} şi $2^{63} - 1$;
- ulong reprezintă tip fără semn pe 64 de biţi, între 0 şi $2^{64} - 1$;
- char reprezintă tip fără semn pe 16 biţi, cu valori între 0 şi 65535. Mulţimea valorilor posibile pentru char corespunde setului de caractere Unicode.

Reprezentarea unei variable de tip întreg se poate face sub formă de şir de cifre zecimale sau hexazecimale, urmate eventual de un prefix. Numerele exprimate în hexazecimal sunt prefixate cu "0x" sau "0X". Regulile după care se asignează un tip pentru o valoare sunt:

1. dacă şirul de cifre nu are un sufix, atunci el este considerat ca fiind primul tip care poate să conţină valoarea dată: int, uint, long, ulong;
2. dacă şirul de cifre are sufixul u sau U, el este considerat ca fiind din primul tip care poate să conţină valoarea dată: uint, ulong;
3. dacă şirul de cifre are sufixul l sau L, el este considerat ca fiind din primul tip care poate să conţină valoarea dată: long, ulong;
4. dacă şirul de cifre are sufixul ul, uL, Ul, UL, lu, lU, Lu, LU, el este considerat ca fiind din tipul ulong.

Dacă o valoare este în afara domeniului lui ulong, apare o eroare la compilare.

Literalii de tip caracter au forma: 'car' unde car poate fi exprimat printr–un caracter, printr–o secvenţă escape simplă, secvenţă escape hexazecimală sau secvenţă escape Unicode. În prima formă poate fi folosit orice caracter exceptând apostrof, backslash şi new line. Secvenţă escape simplă poate fi: `\'`, `\"`, `\\`, `\0`, `\a`, `\b`, `\f`, `\n`, `\r`, `\t`, `\v`, cu semnificaţiile cunoscute din C++. O secvenţă escape hexazecimală începe cu `\x` urmat de 1–4 cifre hexa. Deşi ca reprezentare, char este identic cu ushort, nu toate operaţiile ce de pot efectua cu ushort sunt valabile şi pentru char.

În cazul în care o operaţie aritmetică produce un rezultat care nu poate fi reprezentat în tipul destinaţie, comportamentul depinde de utilizarea operatorilor sau a declaraţiilor *checked* şi *unchecked* (care se pot utiliza în sursă sau din linia de compilare): în context checked, o eroare de depăşire duce la aruncarea unei excepţii de tip System.OverflowException. În context unchecked, eroarea de depăşire este ignorată, iar biţii semnificativi care nu mai încap în reprezentare sunt eliminaţi.

Exemplu:

```
byte i=255;
unchecked
{
  i++;
}//i va avea valoarea 0, nu se semnaleaza eroare
checked
{
  i=255;
  i++;//se va arunca exceptie System.OverflowException
}
```

Pentru expresiile aritmetice care conţin operatorii ++, --, +, - (unar şi binar), *, / şi care nu sunt conţinute în interiorul unui bloc de tip checked, comportamentul este specificat prin intermediul opţiunii /checked[+|−] dat din linia de comandă pentru compilator.

Tipuri în virgulă mobilă

Sunt prezente 2 tipuri numerice în virgulă mobilă: float şi double. Tipurile sunt reprezentate folosind precizie de 32, respectivi 64 de biţi, folosind formatul IEC 60559, care permit reprezentarea valorilor de "0 pozitiv" şi "0 negativ" (se comportă la fel, dar anumite operaţii duc la obţinerea acestor două valori), $+\infty$ şi $-\infty$ (obţinute prin împărţirea unui număr strict pozitiv, respectiv strict negativ la 0), a valorii Not–a–Number (NaN) (obţinută prin operaţii în virgulă mobilă invalide, de exemplu 0/0 sau $\sqrt{-1}$), precum şi un set finit de numere[8]. Tipul float poate reprezenta valori cuprinse între 1.5×10^{-45} şi 3.4×10^{38} (şi din domeniul negativ corespunzător), cu o precizie de 7 cifre. Double poate reprezenta valori cuprinse între 5.0×10^{-324} şi 1.7×10^{308} cu o precizie de 15-16 cifre.

Operaţiile cu floating point nu duc niciodată la apariţia de excepţii, dar ele pot duce, în caz de operaţii invalide, la valori 0, infinit sau NaN.

[8]Pentru detalii vezi [4], pag. 93.

Literalii care specifică un număr reprezentat în virgulă mobilă au forma:
literal–real::
cifre-zecimale . cifre-zecimale exponent$_{optional}$ sufix-de-tip-real$_{optional}$
. cifre-zecimale exponent$_{optional}$ sufix-de-tip-real$_{optional}$
cifre-zecimale exponent sufix-de-tip-real$_{optional}$
cifre-zecimale sufix-de-tip-real,
unde
exponent::
e semn$_{optional}$ cifre-zecimale
E semn$_{optional}$ cifre-zecimale,
semn este + sau -, sufix-de-tip-real este F, f, D, d. Dacă nici un sufix de tip real nu este specificat, atunci literalul dat este de tip double. Sufixul f sau F specifică tip float, d sau D specifică double. Dacă literalul specificat nu poate fi reprezentat în tipul precizat, apare eroare de compilare.

Tipul decimal

Este un tip de date reprezentat pe 128 de biţi, gândit a fi folosit în calcule financiare sau care necesită precizie mai mare. Poate reprezenta valori aflate în intervalul 1.0×10^{-28} şi 7.9×10^{28}, cu 28 de cifre semnificative. Acest tip nu poate reprezenta zero cu semn, infinit sau NaN. Dacă în urma operaţiilor, un număr este prea mic pentru a putea fi reprezentat ca decimal, atunci el este făcut 0, iar dacă este prea mare, rezultă o excepţie. Diferenţa principală faţă de tipurile în virgulă mobilă este că are o precizie mai mare, dar un domeniu de reprezentare mai mic. Din cauza aceasta, nu se fac conversii implicite între nici un tip în virgulă mobilă şi decimal şi nu este posibilă mixarea variabilelor de acest tip într-o expresie, fără conversii explicite.

Literalii de acest tip se exprimă folosind ca sufix-de-tip-real caracterele m sau M. Dacă valoarea specificată nu poate fi reprezentată prin tipul decimal, apare o eroare la compilare.

Tipul bool

Este folosit pentru reprezentarea valorilor de adevăr true şi false. Literalii care se pot folosi sunt true şi false. Nu există conversii standard între bool şi nici un alt tip.

2.2.3 Tipul enumerare

Tipul enumerare este un tip valoare, construit pentru a permite declararea constantelor înrudite, într–o manieră clară şi sigură din punct de vedere al

tipului. Un exemplu este:

```
using System;
public class Draw
{
  public enum LineStyle
  {
    Solid
    Dotted,
    DotDash
  }

  public void DrawLine(int x1, int y1, int x2, int y2,
    LineStyle lineStyle)
  {
    if (lineStyle == LineStyle.Solid)
    {
      //cod desenare linie continua
    }
    else
    if (lineStyle == LineStyle.Dotted)
    {
      //cod desenare linie punctata
    }
    else
    if (lineStyle == LineStyle.DotDash)
    {
      //cod desenare segment linie-punct
    }
    else
    {
      throw new ArgumentException(``Invalid line style'');
    }
  }
}

class Test
{
  public static void Main()
  {
    Draw draw = new Draw();
```

```
    draw.DrawLine(0, 0, 10, 10, Draw.LineStyle.Solid);
    draw.DrawLine(0, 0, 10, 10, (Draw.LineStyle)100);
  }
}
```

Al doilea apel este legal, deoarece valorile care se pot specifica pentru un enum nu sunt limitate la valorile declarate în enum. Ca atare, programatorul trebuie să facă validări suplimentare pentru a determina consistenţa valorilor. În cazul de faţă, la apelul de metodă se aruncă o excepţie (excepţiile vor fi tratate pe larg într-un curs viitor).

Ca şi mod de scriere a enumerărilor, se sugerează folosirea convenţiei Pascal atât pentru numele tipului cât şi pentru numele valorilor conţinute, precum şi evitarea folosirii prefixelor sau sufixelor pentru acestea.

Enumerările nu pot fi declarate abstracte şi nu pot fi derivate. Orice enum este derivat automat din System.Enum, care este la rândul lui derivat din System.ValueType); astfel, metodele moştenite de la tipurile părinte sunt utilizabile de către orice variabilă de tip enum.

Fiecare tip enumerare care este folosit are un tip de reprezentare[9], pentru a se cunoaşte cât spaţiu de memorie trebuie să fie alocat unei variabile de acest tip. Dacă nu se specifică nici un tip de reprezentare (ca mai sus), atunci se presupune implicit tipul int. Specificarea unui tip de reprezentare (care poate fi orice tip integral, exceptând tipul char) se face prin enunţarea tipului după numele enumerării:

```
enum MyEnum : byte
{
  small,
  large
}
```

Specificarea este folosită atunci când dimensiunea în memorie este importantă, sau când se doreşte crearea unui tip de indicator (un tip flag) al cărui număr de stări diferă de numărul de biţi alocaţi tipului int (modelare de flag-uri):

```
enum ActionAttributes : ulong
{
Read = 1,
Write = 2,
Delete = 4,
Query = 8,
```

[9]Engl: underlying type

```
Sync = 16
//etc
}
...
ActionAttributes aa=ActionAttributes.Read|ActionAttributes.Write
| ActionAttributes.Query;
...
```

În mod implicit, valoarea primului membru al unei structuri este 0, şi fiecare variablă care urmează are valoarea mai mare cu o unitate decât precedenta. La dorinţă, valoarea fiecărei variabile poate fi specificată explicit:

```
enum Values
{
  a = 1,
  b = 2,
  c = a + b
}
```

Următoarele observaţii se impun relativ la lucrul cu tipul enumerare:

1. valorile specificate ca iniţializatori trebuie să fie reprezentabile prin tipul de reprezentare a enumerării, altfel apare o eroare la compilare:

   ```
   enum Out : byte
   {
     A = -1
   }//eroare semnalata la compilare
   ```

2. mai mulţi membri pot avea aceeaşi valoare (manevră dictată de semantica tipului construit):

   ```
   enum ExamState
   {
     passed = 10,
     failed = 1,
     rejected = failed
   }
   ```

3. dacă pentru un membru nu este dată o valoare, acesta va lua valoarea membrului precedent + 1 (cu excepţia primului membru – vezi mai sus)

4. nu se permit referinţe circulare:

```
enum CircularEnum
{
  A = B,
  B
}//A depinde explicit de B, B depinde implicit de A
//eroare semnalata la compilare
```

5. este recomandat ca orice tip enum să conţină un membru cu valoarea 0, pentru că în anumite contexte valoarea implicită pentru o variabilă enum este 0, ceea ce poate duce la inconsistenţe şi bug-uri greu de depanat

```
enum Months
{
  InvalidMonth,//are valoarea implicita 0, fiind primul element
  January,
  February,
  //etc
}
```

Tipurile enumerare pot fi convertite către tipul lor de bază şi înapoi, folosind o conversie explicită (cast):

```
enum Values
{
  a = 1,
  b = 5,
  c= 3
}

class Test
{
  public static void Main()
  {
    Values v = (Values)3;
    int ival = (int)v;
  }
}
```

Valoarea 0 poate fi convertită către un enum fără cast:

```
...
MyEnum me;
...
if (me == 0)
{
  //cod
}
```

Următorul cod arată câteva din artificiile care pot fi aplicate tipului enumerare: obţinerea tipului unui element de tip enumerare precum şi a tipului clasei de bază, a tipului de reprezentare, a valorilor conţinute (ca nume simbolice şi ca valori), conversie de la un string la un tip enumerare pe baza numelui, etc. Exemplul este preluat din <FrameworkSDK> \Samples\Technologies\ValueAndEnumTypes.

```
using System;

namespace DemoEnum
{
class DemoEnum
{
enum Color
{
Red   = 111,
Green = 222,
Blue  = 333
}
private static void DemoEnums()
{
Console.WriteLine("\n\nDemo start: Demo of enumerated types.");
Color c = Color.Red;

// What type is this enum & what is it derived from
Console.WriteLine("   The " + c.GetType() + " type is derived from "
 + c.GetType().BaseType);

// What is the underlying type used for the Enum's value
Console.WriteLine("   Underlying type: " + Enum.GetUnderlyingType(
             typeof(Color)));

// Display the set of legal enum values
```

```
Color[] o = (Color[]) Enum.GetValues(c.GetType());
Console.WriteLine("\n   Number of valid enum values: " + o.Length);
for (int x = 0; x < o.Length; x++)
{
Color cc = ((Color)(o[x]));
Console.WriteLine("   {0}: Name={1,7}\t\tNumber={2}", x,
cc.ToString("G"), cc.ToString("D"));
}

// Check if a value is legal for this enum
Console.WriteLine("\n   111 is a valid enum value: " + Enum.IsDefined(
          c.GetType(), 111));   // True
Console.WriteLine("   112 is a valid enum value: " + Enum.IsDefined(
    c.GetType(), 112));   // False

// Check if two enums are equal
Console.WriteLine("\n   Is c equal to Red: " + (Color.Red == c));//True
Console.WriteLine("   Is c equal to Blue: "  + (Color.Blue == c));//False

// Display the enum's value as a string using different format specifiers
Console.WriteLine("\n   c's value as a string: " + c.ToString("G"));//Red
Console.WriteLine("   c's value as a number: " + c.ToString("D"));//111

// Convert a string to an enum's value
c = (Color) (Enum.Parse(typeof(Color), "Blue"));
try
{
c = (Color) (Enum.Parse(typeof(Color), "NotAColor"));//Not valid,
//raises exception
}
catch (ArgumentException)
{
Console.WriteLine("   'NotAColor' is not a valid value for this enum.");
}

// Display the enum's value as a string
Console.WriteLine("\n   c's value as a string: " + c.ToString("G"));//Blue
Console.WriteLine("   c's value as a number: " + c.ToString("D"));//333

Console.WriteLine("Demo stop: Demo of enumerated types.");
}
```

```
static void Main()
{
DemoEnums();
}
}
}
```

2.3 Tablouri

În cele mai multe cazuri nu se cunoaşte aprioric numărul de obiecte care se vor folosi la momentul rulării. O soluţie pentru stocarea elementelor având acelaşi tip o reprezintă tablourile. Sintaxa de declarare este asemănătoare cu cea din Java sau C++, dar fiecare tablou este un obiect, derivat din clasa abstractă System.Array. Accesul la elemente se face prin intermediul indicilor care încep de la 0 şi se termină la numărul de elemente-1; orice depăşire a indicilor duce la apariţia unei excepţii: System.IndexOutOfRangeException. O variabilă de tip tablou poate avea valoare de null sau poate să indice către o instanţă validă.

2.3.1 Tablouri unidimensionale

Declararea unui tablou unidimensional se face prin plasarea de paranteze drepte între numele tipului tabloului şi numele său, ca mai jos[10]:

```
int[] sir;
```

Declararea de mai sus nu duce la alocare de spaţiu pentru memorarea şirului; *instaţierea* se poate face ca mai jos:

```
sir = new int[10];
```

Exemplu:

```
using System;
class Unidimensional
{
  public static int Main()
  {
    int[] sir;
    int n;
```

[10]Spre deosebire de Java, nu se poate modifica locul parantezelor, adică nu se poate scrie: int sir[].

```
    Console.Write(‘‘Dimensiunea vectorului: ’’);
    n = Int32.Parse( Console.ReadLine() );
    sir = new int[n];
    for( int i=0; i<sir.Length; i++)
    {
      sir[i] = i * i;
    }
    for( int i=0; i<sir.Length; i++)
    {
      Console.WriteLine(‘‘sir[{0}]={1}’’, i, sir[i]);
    }
    return 0;
  }
}
```

În acest exemplu se foloseşte proprietatea[11] Length, care returnează numărul *tuturor* elementelor vectorului (lucru vizibil la tablourile multidimensionale rectangulare). De menţionat că în acest context n şi *sir* nu se pot declara la un loc, adică declaraţii de genul `int[] sir, n;` sau `int n, []sir;` sunt incorecte (prima este corectă din punct de vedere sintactic, dar ar rezulta că n este şi el un tablou; în al doilea caz, declaraţia nu este corectă sintactic).

Se pot face iniţializări ale valorilor conţinute într–un tablou:

```
int[] a = new int[] {1,2,3};
```

sau în forma mai scurtă:

```
int[] a = {1,2,3};
```

2.3.2 Tablouri multidimensionale

C# cunoaşte două tipuri de tablouri multidimensionale: rectangulare şi neregulate[12]. Numele lor vine de la forma pe care o pot avea.

Tablouri rectangulare

Tablourile rectangulare au proprietatea că pe fiecare dimensiune se află acelaşi număr de elemente. Se declară ca în exemplul de mai jos:

```
int[,] tab;
```

[11] Pentru noţiunea de proprietate, vezi la partea despre clase.

[12] Engl: jagged arrays.

unde tab este un tablou rectangular bidimensional. Instanţierea se face:

```
tab = new int[2,3];
```

rezultând un tablou cu 2 linii şi 3 coloane. Referirea la elementul aflat pe linia i şi coloana j se face cu $tab[i, j]$.

La declararea tabloului se poate face şi iniţializare:

```
int[,] tab = new int[,] {{1,2},{3,4}};
```

sau, mai pe scurt:

```
int[,] tab = {{1, 2}, {3, 4}};
```

Exemplu:

```
using System;
class Test
{
  public static void Main()
  {
    int[,] tabInm = new int[10,10];
    for( int i=0; i<tabInm.GetLength(0); i++ )
    {
      for( int j=0; j<tabInm.GetLength(1); j++)
      {
        tabInm[i,j] = i * j;
      }
    }
    for( int i=0; i<tabInm.GetLength(0); i++)
    {
      for( int j=0; j<tabInm.GetLength(1); j++)
      {
        Console.WriteLine(``{0}*{1}={2}'', i, j, tabInm[i,j]);
      }
    }
    Console.WriteLine(``tabInm.Length={0}'', tabInm.Length);
  }
}
```

După ce se afişează tabla înmulţirii până la 10, se va afişa: `tabInm.Length=100`, deoarece proprietatea Length dă numărul total de elemente aflat în tablou (pe toate dimensiunile). Am folosit însă metoda `GetLength(d)` care returnează

numărul de elemente aflate pe dimensiunea numărul d (numărarea dimensiunilor începe cu 0).

Determinarea numărului de dimensiuni pentru un tablou rectangular la run–time se face folosind proprietatea Rank a clasei de bază System.Array.

Exemplu:

```
using System;
class Dimensiuni
{
  public static void Main()
  {
    int[] t1 = new int[2];
    int[,] t2 = new int[3,4];
    int[,,] t3 = new int[5,6,7];
    Console.WriteLine(‘‘t1.Rank={0}\nt2.Rank={1}\nt3.Rank={2}’’,
    t1.Rank, t2.Rank, t3.Rank);
  }
}
```

Pe ecran va apărea:

```
t1.Rank=1
t2.Rank=2
t3.Rank=3
```

Tablouri neregulate

Un tablou neregulat[13] reprezintă un tablou de tabouri. Declararea unui tablou neregulat cu două dimensiuni se face ca mai jos:

```
int[][] tab;
```

Referirea la elementul de indici i şi j se face prin `tab[i][j]`.

Exemplu:

```
using System;
class JaggedArray
{
  public static void Main()
  {
    int[][] a = new int[2][];
    a[0] = new int[2];
```

[13]Engl: jagged array

```
    a[1] = new int[3];
    for( int i=0; i<a[0].Length; i++)
    {
      a[0][i] = i;
    }
    for( int i=0; i<a[1].Length; i++)
    {
      a[1][i] = i * i;
    }
    for(int i=0; i<a.Length; i++)
    {
      for( int j=0; j<a[i].Length; j++ )
      {
        Console.Write(``{0} '', a[i][j]);
      }
      Console.WriteLine();
    }
    Console.WriteLine(``a.Rank={0}'', a.Rank);
  }
}
```

va scrie pe ecran:

```
0 1
0 1 4
a.Rank=1
```

Ultima linie afişată se explică prin faptul că un tablou neregulat este un vector care conţine referinţe, deci este unidimensional.

Iniţializarea valorilor unui tablou neregulat se poate face la declarare:

```
int[][] myJaggedArray = new int [][]
                        {
                           new int[] {1,3,5,7,9},
                           new int[] {0,2,4,6},
                           new int[] {11,22}
                        };
```

Forma de mai sus se poate prescurta la:

```
int[][] myJaggedArray = {
                           new int[] {1,3,5,7,9},
                           new int[] {0,2,4,6},
                           new int[] {11,22}
                        };
```

2.4 Şiruri de caractere

Tipul de date folosit pentru reprezentarea şirurilor de caractere este clasa System.String (pentru care se poate folosi aliasul "string"; reamintim că este un tip predefinit). Obiectele de acest tip sunt imutabile (caracterele conţinute nu se pot schimba, dar pe baza unui şir se poate obţine un alt şir). Şirurile pot conţine secvenţe escape şi pot fi de două tipuri: regulate şi de tip "verbatim"[14]. Şirurile regulate sunt demarcate prin ghilimele şi necesită secvenţe escape pentru reprezentarea caracterelor escape.

Exemplu:

```
String a = "string literal";
String versuri = "vers1\nvers2";
String caleCompleta = "\\\\minimax\\protect\\csharp";
```

Pentru situaţia în care se utilizează masiv secvenţe escape, se pot folosi şirurile verbatim. Un literal de acest tip are simbolul "@" înaintea ghilimelelor de început. Pentru cazul în care ghilimelele sunt întâlnite în interiorul şirului, ele se vor dubla. Un şir de caractere poate fi reprezentat pe mai multe rânduri fără a folosi caracterul `\n`. Şirurile verbatim sunt folosite pentru a face referiri la fişiere sau chei în regiştri, sau pentru prelucrarea fişierelor.

Exemple:

```
String caleCompleta=@"\\minimax\protect\csharp";
//ghilimelele se dubleaza intr-un verbatim string
String s=@"notiunea ""aleator"" se refera...";
//string multilinie reprezentat ca verbatim
String dialog=@"-Alo? Cu ce va ajutam?
-As avea nevoie de o informatie.";
```

Operatorii "==" şi "!=" pentru două şiruri de caractere se comportă în felul următor: două şiruri de caractere se consideră egale dacă sunt fie amândouă null, fie au aceeaşi lungime şi caracterele de pe aceleaşi poziţii coincid; "!=" dă negarea relaţiei de egalitate. Pentru a se compara două şiruri din punct de vedere al referinţelor, trebuie ca măcar unul dintre obiecte să fie convertit la object:

```
class Test
{
  public static void Main()
  {
```

[14]Engl: verbatim literals

```
    string a = "Salut", b = a.Clone();
    System.Console.WriteLine("a==b: {0}", a==b);
    System.Console.WriteLine("(object)a==b: {0}",
      (object)a==b);
  }
}
```

La dispozitivul standard de ieşire se va afişa:

```
a==b: true
(object)a==b: false
```

Alt operator care se poate utiliza este "+", reprezentând concatenarea:

```
String s = "Salut" + " " + "lume";
s += "!!!!";
```

Clasa String pune la dispoziţie metode pentru: comparare (Compare, CompareOrdinal, CompareTo), căutare (EndsWith, StartsWith, IndexOf, LastIndexOf), modificare (a se înţelege obţinerea altor obiecte pe baza celui curent - Concat, CopyTo, Insert, Join, PadLeft, PadRight, Remove, Replace, Split, Substring, ToLower, ToUpper, Trim, TrimEnd, TrimStart)[15]. Accesarea unui caracter aflat pe o poziţie i a unui şir s se face prin folosirea parantezelor drepte, cu aceleaşi restricţii asupra indicelui ca şi pentru tablouri: `s[i]`.

În clasa object se află metoda ToString() care este suprascrisă în fiecare clasă ale cărei instanţe pot fi tipărite. Pentru obţinerea unei reprezentări diferite se foloseşte metoda `String.Format()`.

Un exemplu de folosire a funcţiei `Split()` pentru despărţirea unui şir în funcţie de separatori este:

```
using System;
class Class1
{
  static void Main(string[] args)
  {
    String s = "Oh, I hadn't thought of that!";
    char[] x = {' ', ','};
    String[] tokens = s.Split( x );
    for(int i=0; i<tokens.Length; i++)
    {
      Console.WriteLine("Token: {0}", tokens[i]);
```

[15] A se vedea exemplele din MSDN.

```
    }
  }
}
```

va afişa pe ecran:

```
Token: Oh
Token:
Token: I
Token: hadn't
Token: thought
Token: of
Token: that!
```

De remarcat că pentru caracterul apostrof nu este obligatorie secvenţa escape în cazul şirurilor de caractere. Al doilea lucru care trebuie explicat este că al doilea token este cuvântul vid, care apare între cei doi separatori alăturaţi: virgula şi spaţiul. Metoda Split() nu face gruparea mai multor separatori, lucru care ar fi de dorit în prezenţa a doi separatori alăturaţi. Pentru aceasta se folosesc expresii regulate.

Pentru a lucra cu şiruri de caractere care permit modificarea lor se foloseşte clasa StringBuilder, din namespace–ul System.Text.

2.4.1 Expresii regulate

În cazul în care funcţiile din clasa String nu sunt suficient de puternice, namespace–ul System.Text.RegularExpresions pune la dispoziţie o clasă de lucru cu expresii regulate numită Regex. Expresiile regulate reprezintă o metodă extrem de facilă de a opera căutări/înlocuiri pe text. Forma expresiilor regulate este cea din limbajul Perl.

Această clasă foloseşte o tehnică interesantă pentru mărirea performanţelor: dacă programatorul vrea, se scrie o secvenţă "din mers" pentru a implementa potrivirea expresiei regulate, după care codul este rulat[16].

Exemplul anterior poate fi rescris corect din puct de vedere al funcţionalităţii prin folosirea unei expresii regulate, pentru a prinde şi cazul separatorilor multipli adiacenţi:

```
class ExpresieRegulata
{
  static void Main(string[] args)
  {
```

[16]Codul este scris direct în IL.

```
    String s = "Oh, I hadn't thought of that!";
    //separator: virgula, spatiu sau punct si virgula
    //unul sau mai multe, orice combinatie
    Regex regex = new Regex("[, ;]+");
    String[] strs = regex.Split(s);
    for( int i=0; i<strs.Length; i++)
    {
      Console.WriteLine("Word: {0}", strs[i]);
    }
}
```

care va produce:

```
Word: Oh
Word: I
Word: hadn't
Word: thought
Word: of
Word: that!
```

Curs 3

Clase – generalităţi. Instrucţiuni. Spaţii de nume

3.1 Clase – vedere generală

Clasele reprezintă tipuri referinţă definite de utilizator. O clasă poate să moştenească o singură clasă şi poate implementa mai multe interfeţe.

Clasele pot conţine constante, câmpuri, metode, proprietăţi, evenimente, indexatori, operatori, constructori de instanţă, destructori, constructori de clasă, tipuri imbricate. Fiecare membru poate conţine un nivel de accesare, care controlează gradul de acces la el. O descriere este dată în tabelul 3.1:

Tabelul 3.1: Modificatori de acces ai membrilor unei clase

Accesor	**Semnificaţie**
public	Acces nelimitat
protected	Acces limitat la clasa conţinătoare sau la tipuri derivate din ea
internal	Acces limitat la acest assembly
protected internal	Acces limitat la acest assembly sau la tipuri derivate din clasă
private	Acces limitat la clasă; modificatorul implicit de acces

```
using System;
```

```
class MyClass
{
  public MyClass()
  {
    Console.WriteLine("Constructor instanta");
  }
  public MyClass( int value )
  {
    MyField = value;
    Console.WriteLine("Constructor instanta");
  }
  public const int MyConst = 12;
  public int MyField = 34;
  public void MyMethod()
  {
    Console.WriteLine("MyClass.MyMethod");
  }
  public int MyProperty
  {
    get
    {
      return MyField;
    }
    set
    {
      MyField = value;
    }
  }
  public int this[int index]
  {
    get
    {
      return 0;
    }
    set
    {
      Console.WriteLine("this[{0}]={1}", index, value);
    }
  }
  public event EventHandler MyEvent;
  public static MyClass operator+(MyClass a, MyClass b)
```

```
  {
    return new MyClass(a.MyField + b.MyField);
  }
}

class Test
{
  static void Main()
  {
    MyClass a = new MyClass();
    MyClass b = new MyClass(1);
    Console.WriteLine("MyConst={0}", MyClass.MyConst);
    a.MyField++;
    Console.WriteLine("a.MyField={0}", a.MyField);
    a.MyMethod();
    a.MyProperty++;
    Console.WriteLine("a.MyProperty={0}", a.MyProperty);
    a[3] = a[1] = a[2];
    Console.WriteLine("a[3]={0}", a[3]);
    a.MyEvent += new EventHandler(MyHandler);
    MyClass c = a + b;
  }

  static void MyHandler(object Sender, EventArgs e)
  {
    Console.WriteLine("Test.MyHandler");
  }

  internal class MyNestedType
  {}
}
```

Constanta este un membru al unei clase care reprezintă o valoare constantă, care poate fi evaluată la compilare. Constantele pot depinde de alte constante, atâta timp cât nu se creează dependenţe circulare. Ele sunt considerate automat membri statici (dar este interzis să se folosească specificatorul static în faţa lor). Ele pot fi accesate exclusiv prin intermediul numelui de clasă (MyClass.MyConst), şi nu prin intermediul vreunei instanţe (a.MyConst).

Câmpul este un membru reprezentând o variabilă asociată unui obiect.

Metoda este un membru care implementează un calcul sau o acţiune care poate fi efectuată asupra unui obiect sau asupra unei clase. Metodele statice (care au în antet cuvântul cheie static) sunt accesate prin intermediul numelui de clasă, pe când cele nestatice (metode instanţă) sunt apelate prin intermediul unui obiect.

Proprietatea este un membru care dă acces la o caracteristică a unui obiect sau unei clase. Exemplele folosite până acum includeau lungimea unui vector, numărul de caractere ale unui şir de caractere, etc. Sintaxa pentru accesara câmpurilor şi a proprietăţilor este aceeaşi. Reprezintă o altă modalitate de implementare a accesorilor pentru obiecte.

Evenimentul este un membru care permite unei clase sau unui obiect să pună la dispoziţia altora notificări asupra evenimentelor. Tipul acestei declaraţii trebuie să fie un tip delegat. O instanţă a unui tip delegat încapsulează una sau mai multe entităţi apelabile. Exemplu:

```
public delegate void EventHandler(object sender,
                                  System.EventArgs e);
public class Button
{
  public event EventHandler Click;
  public void Reset()
  {
    Click = null;
  }
}

using System;
public class Form1
{
  Button Button1 = new Button1();

  public Form1()
  {
    Button1.Click += new EventHandler(Button1_Click);
  }

  void Button1_Click(object sender, EventArgs e )
  {
    Console.WriteLine("Button1 was clicked!");
```

```
  }

  public void Disconnect()
  {
    Button1.Click -= new EventHandler(Button1_Click);
  }
}
```

Mai sus clasa Form1 adaugă Button1_Click ca tratare de eveniment[1] pentru evenimentul Click al lui Button1. În metoda Disconnect(), acest event handler este înlăturat.

Operatorul este un membru care defineşte semnificaţia (supraîncărcarea) unui operator care se aplică instanţelor unei clase. Se pot supraîncărca operatorii binari, unari şi de conversie.

Indexatorul este un membru care permite unui obiect să fie indexat în acelaşi mod ca un tablou (pentru programatorii C++: supraîncărcarea operatorului []).

Constructorii instanţă sunt membri care implementează acţiuni cerute pentru iniţializarea fiecărui obiect.

Destructorul este un membru special care implementează acţiunile cerute pentru a distruge o instanţă a unei clase. Destructorul nu are parametri, nu poate avea modificatori de acces, nu poate fi apelaţi explicit şi poate fi apelat automat de către garbage collector.

Constructorul static este un membru care implementează acţiuni necesare pentru a iniţializa o clasă, mai exact membrii statici ai clasei. Nu pot avea parametri, nu pot avea modificatori de acces, nu sunt apelaţi explicit, ci automat de către sistem.

Moştenirea este de tip simplu, iar rădăcina ierarhiei este clasa object (alias System.Object).

3.2 Transmiterea de parametri

În general, transmiterea parametrilor se face prin valoare. Acest lucru înseamnă că la apelul unei metode în stiva gestionată de compilator se copiază

[1]Engl: event handler

valoarea parametrului actual transmis, iar la revenire din metodă această valoare va fi ştearsă. Exemplificăm mai jos acest lucru pentru tipurile valoare şi referinţă.

```
using System;
class DemoTipValoare
{
  static void f(int x)
  {
    Console.WriteLine("la intrare in f: {0}", x );
    ++x;
    Console.WriteLine("la iesire din f: {0}", x );
  }
  static void Main()
  {
    int a = 100;
    Console.WriteLine("inainte de intrare in f: {0}", a);
    f( a );
    Console.WriteLine("dupa executarea lui f: {0}", a);
  }
}
```

Executarea acestui program va avea ca rezultat:

```
inainte de intrare in f: 100
la intrare in f: 100
la iesire din f: 101
dupa executarea lui f: 100
```

Pentru variable de tip referinţă, pe stivă se depune tot o copie a valorii obiectului. Însă pentru un asemenea tip de variabilă acest lucru înseamnă că pe stivă se va depune ca valoare adresa de memorie la care eset stocat obiectul respectiv. Ca atare, metoda apelată poate să modifice starea obiectului care se transmite:

```
class Employee
{
  public String name;
  public decimal salary;
}

class Test
```

```
{
  static void Main()
  {
    Employee e = new Employee();
    e.name = "Ionescu";
    e.salary = 300M;
    System.Console.WriteLine("pre: name={0}, salary={1}",
                        e.name, e.salary );
    Method( e );
    System.Console.WriteLine("post: name={0}, salary={1}",
                        e.name, e.salary );
  }

  static void Method( Employee e )
  {
    e.salary += 100;
  }
}
```

va avea ca rezultat:

```
pre: name=Ionescu, salary=300
post: name=Ionescu, salary=400
```

Totuşi, chiar şi în cazul tipului referinţă încercarea de a re–crea în interiorul unei metode un obiect transmis ca parametru nu are nici un efect după terminarea ei:

```
class MyClass
{
  public int x;
}

class Test
{
  static void f(MyClass myClass)
  {
    Console.WriteLine("intrare in f: {0}", myClass.x);
    myClass = new MyClass();
    myClass.x = -100;
    Console.WriteLine("iesire din f: {0}", myClass.x);
  }
```

```
  static void Main()
  {
    MyClass myClass = new MyClass();
    myClass.x = 100;
    Console.WriteLine("inainte de apel: {0}", myClass.x);
    f( myClass );
    Console.WriteLine("dupa apel: {0}", myClass.x);
}
}
```

Ieşirea acestui program va fi:

```
inainte de apel: 100
intrare in f: 100
iesire din f: -100
dupa apel: 100
```

Există situaţii în care acest comportament nu este cel dorit: am vrea ca efectul asupra unui parametru să se menţină şi după ce metoda apelată s–a terminat.

Un parametru *referinţă* este folosit tocmai pentru a rezolva problema transmiterii prin valoare, folosind referinţă (un alias) pentru entitatea dată de către metoda apelantă. Pentru a transmite un parametru prin referinţă, se prefixează cu cuvântul cheie ref la apel sau la declarare de metodă:

```
using System;
class Test
{
  static void Swap( ref int a, ref int b)
  {
    int t = a;
    a = b;
    b = t;
  }

  static void Main()
  {
    int x=1, y=2;
    Console.WriteLine("inainte de apel: x={0}, y={1}", x, y);
    Swap( ref a, ref b )
    Console.WriteLine("dupa apel: x={0}, y={1}", x, y);
  }
}
```

va realiza interschimbarea valorilor a şi b.

Una din trăsăturile specifice parametrilor referinţă este că valorile pentru care se face apelul trebuie să fie iniţializate. Neasignarea de valori pentru x şi y în exemplul de mai sus duce o eroare de compilare. Mai clar, exemplul de mai jos genereaza eroare la compilare, mesajul fiind: "Use of unassigned local variable 'x'".

```
class TestRef
{
  static void f(ref int x)
  {
    x = 100;
  }
  static void Main()
  {
    int x;
    f(ref x);
  }
}
```

Există cazuri în care dorim să obţinem acelaşi efect ca la parametrii referinţă, dar fără a trebui să iniţializăm argumentele date de către metoda apelantă (de exemplu când valoarea acestui parametru se calculează în interiorul metodei apelate). Pentru aceasta există parametrii de ieşire[2], similar cu parametrii referinţă, cu deosebirea că nu trebuie asignată o valoare parametrului de apel:

```
using System;
class Test
{
  static void Main()
  {
    int l = 10;
    double area;
    ComputeSquareArea( l, out area);
    Console.WriteLine("Area is: {0}", area);
  }

  static void ComputeSquareArea( double l, out double area )
  {
```

[2]Engl: output parameters

```
    area = l * l;
  }
}
```

Pentru toate tipurile de parametri de mai sus există o mapare 1 la 1 între parametrii actuali şi cei formali. Un parametru vector[3] permite o relaţie de tipul unul-la-mulţi: mai mulţi parametri actuali pot fi referite prin intermediul unui singur parametru formal. Un astfel de parametru se declară folosind modificatorul params. Pentru o implementare de metodă, putem avea cel mult un parametru de tip vector şi acesta trebuie să fie ultimul în lista de parametri. Acest parametru formal este tratat ca un tablou unidimensional:

```
using System;
class Test
{
  static void F(params int[] args)
  {
    Console.WriteLine("# of parameters: {0}", args.Length);
    for( int i=0; i<args.Length; i++)
    {
      Console.WriteLine("args[{0}]={1}", i, args[i]);
    }
  }

  static void Main()
  {
    F();
    F(1);
    F(1,2);
    F(new int[] {1,2,3});
  }
}
```

Acest tip de transmitere se foloseşte şi de către metoda WriteLine (sau Write) a clasei Console, i.e. există în această clasă o metodă de tipul:

```
public static void WriteLine(string format, params object[] args){...}
```

[3]Engl: parameter array

3.3 Conversii

O conversie permite ca o expresie de un anumit tip să fie tratată ca fiind de alt tip. Conversiile pot fi implicite sau explicite, aceasta specificând de fapt dacă un operator de cast (conversie) este sau nu necesar.

3.3.1 Conversii implicite

Sunt clasificate ca şi conversii implicite următoarele:

- conversiile identitate
- conversiile numerice implicite
- conversiile implicite de tip enumerare
- conversiile implicite de referinţe
- boxing
- conversiile implicite ale expresiilor constante
- conversii implicite definite de utilizator

Conversiile implicite pot apărea într–o varietate de situaţii, de exemplu apeluri de funcţii sau atribuiri. Conversiile implicite predefinite nu determină niciodată apariţia de excepţii.

Conversiile indentitate

O conversie identitate converteşte de la un tip oarecare către acelaşi tip.

Conversiile numerice implicite

Conversiile numerice implicite sunt:

- de la sbyte la short, int, long, float, double, decimal;
- de la byte la short, ushort, int, uint, long, ulong, float, double, decimal;
- de la short la int, long, double, decimal;
- de la ushort la int, uint, long, ulong, float, double, decimal;
- de la int la long, float, double, decimal;

- de la uint la long, ulong, float, double, decimal;
- de la long la float, double, decimal;
- de la ulong la float, double, decimal;
- de la char la ushort, int, uint, long, ulong, float, double, decimal;
- de la float la double.

Conversiile de la int, uint, long, ulong la float, precum şi cele de la long sau ulong la double pot duce la o pierdere a preciziei, dar niciodată la o reducere a ordinului de mărime. Alte conversii numerice implicite niciodată nu duc la pierdere de informaţie.

Conversiile de tip enumerare implicite

O astfel de conversie permite ca literalul 0 să fie convertit la orice tip enumerare (chiar dacă acesta nu conţine valoarea 0) - a se vedea 2.2.3, pag. 29.

Conversii implicite de referinţe

Conversiile implicite de referinţe implicite sunt:

- de la orice tip referinţă la object;
- de la orice tip clasă B la orice tip clasă A, dacă B este derivat din A;
- de la orice tip clasă A la orice interfaţă B, dacă A implementează B;
- de al orice interfaţă A la orice interfaţă B, dacă A este derivată din B;
- de la orice tip tablou A cu tipul A_E la un tip tablou B având tipul B_E, cu următoarele condiţii:
  1. A şi B au acelaşi număr de dimensiuni;
  2. atât A_E cât şi B_E sunt tipuri referinţă;
  3. există o conversie implicită de tip referinţă de la A_E la B_E
- de la un tablou la System.Array;
- de la tip delegat la System.Delegate;
- de la orice tip tablou sau tip delegat la System.ICloneable;
- de la tipul null la orice variabilă de tip referinţă;

Conversie de tip boxing

Permite unui tip valoare să fie implicit convertit către tipul object sau System.ValueType sau către orice tip interfaţă pe care tipul valoare îl implementează. O descriere mai amănunţită este dată în secţiunea 3.3.4.

3.3.2 Conversiile implicite ale expresiilor constante

Permit următoarele tipuri de conversii:

- o expresie constantă de tip int poate fi convertită către tipurile sbyte, byte, short, ushort, uint, ulong, cu condiţia ca valoarea expresiei constante să se afle în domeniul tipului destinaţie;
- o expresie constantă de tip long poate fi convertită la tipul ulong, dacă valoarea ce se converteşte nu este negativă.

Conversii implicite definite de utilizator

Constau într–o conversie implicită standard opţională, urmată de execuţia unui operator de conversie implicită utilizator urmată de altă conversie implicită standard opţională. Regulile exacte sunt descrise în [4].

3.3.3 Conversii explicite

Următoarele conversii sunt clasificate ca explicite:

- toate conversiile implicite
- conversiile numerice explicite
- conversiile explicite de enumerări
- conversiile explicite de referinţe
- unboxing
- conversii explicite definite de utilizator

Din cauză că orice conversie implicită este de asemenea şi una explicită, aplicarea operatorului de cast este redundantă:

```
int x = 0;
long y = (long)x;//(long) este redundant
```

Conversii numerice explicite

Sunt conversii de la orice tip numeric la un alt tip numeric pentru care nu există conversie numerică implicită:

- de la sbyte la byte, ushort, uint, ulong, char;
- de la byte la sbyte, char;
- de la short la sbyte, byte, ushort, uint, ulong, char;
- de la ushort la sbyte, byte, short, char;
- de la int la sbyte, byte, short, ushort, int, char;
- de la uint la sbyte, byte, short, ushort, int, uint, long, ulong, char;
- de la long la sbyte, byte, short, ushort, int, uint, ulong, char;
- de la ulong la sbyte, byte, short, ushort, int, uint, long, char;
- de la char la sbyte, byte, short;
- de la float la sbyte, byte, short, ushort, int, uint, long, ulong, decimal;
- de la double la sbyte, byte, short, ushort, int, uint, long, ulong, char, float, decimal;
- de la decimal la sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double;

Pentru că în astfel de conversii pot apărea pierderi de informaţie, există două contexte în care se fac aceste conversii: checked şi unchecked.

În context checked, conversia se face cu succes dacă valoarea care se converteşte este reprezentabilă de către tipul către care se face conversia. În cazul în care conversia nu se poate face cu succes, se va arunca excepţia System.OverflowException. În context unchecked, conversia se face întotdeauna, dar se poate ajunge la pierdere de informaţie sau la valori ce nu sunt bine nedefinite (vezi [4], pag. 115–116).

Conversii explicite de enumerări

Conversiile explicite de enumerări sunt:

- de la sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal la orice tip enumerare;

- de la orice tip enumerare la sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal;
- de la orice tip enumerare la orice tip enumerare.

Conversiile de tip enumerare se fac prin tratarea fiecărui tip enumerare ca fiind tipul întreg de reprezentare, după care se efectuează o conversie implictă sau explicită între tipuri (ex: dacă se doreşte conversia de la un tip enumerare E care are tipul de reprezentare int la un tip byte, se va face o conversie explicită de la int la byte; invers, se va face o conversie implictă de la byte la int).

Conversii explicite de referinţe

Conversiile explicite de referinţe sunt:

- de la object la orice tip referinţă;
- de la orice tip clasă A la orice tip clasă B, cu condiţia ca A să fie clasă de bază pentru B;
- de la orice tip clasă A la orice tip interfaţă B, dacă A nu este nederivabilă şi A nu implementează pe B;
- de la orice tip interfaţă A la orice tip clasă B, dacă B nu este nederivabilă sau cu condiţia ca B să implementeze A;
- de la orice tip interfaţă A la orice tip interfaţă B, dacă A nu este derivat din B;
- de la un tip tablou A cu elemente de tip A_E la un tip tablou B cu elemente B_E, cu condiţiile:
  1. A şi B au acelaşi număr de dimensiuni;
  2. A_E şi B_E sunt tipuri referinţă;
  3. există o conversie de referinţă explicită de la A_E al B_E
- de la System.Array şi interfeţele pe care le implementează la orice tip tablou;
- de la System.Delegate şi interfeţele pe care le implementează la orice tip delegat.

Acest tip de conversii cer verificare la run–time. Dacă o astfel de conversie eşuează, se va arunca o excepţie de tipul System.InvalidCastException.

Unboxing

Unboxing-ul permite o conversie explicită de la object sau System.ValueType la orice tip valoare, sau de la orice tip interfaţă la orice tip valoare care implementează tipul interfaţă. Mai multe detalii se vor da în secţiunea 3.3.4.

Conversii explicite definite de utilizator

Constau într–o conversie standard explicită opţională, urmată de execuţia unei conversii explicite, urmată de o altă conversie standard explicită opţională.

3.3.4 Boxing şi unboxing

Boxing–ul şi unboxing–ul reprezintă modalitatea prin care C# permite utilizarea simplă a sistemului unificat de tipuri. Spre deosebire de Java, unde există tipuri primitive (care nu pot conţine metode) şi tipuri referinţă, în C# toate tipurile sunt derivate din clasa object (alias System.Object). De exemplu, tipul int (alias System.Int32) este derivat din clasa System.ValueType care la rândul ei este derivată din clasa object (alias System.Object). Ca atare, orice int este compatibil cu object.

Boxing

Conversia de tip boxing permite oricărui tip valoare să fie implicit convertit către tipul object sau către un tip interfaţă implementat de tipul valoare. Boxing–ul unei valori constă în alocarea unei variabile de tip obiect şi copierea valorii iniţiale în acea instanţă.

Procesul de boxing al unei valori sau variabile de tip valoare se poate înţelege ca o simulare de creare de clasă pentru acel tip:

```
sealed class T_Box
{
  T value;
  public T_Box(T t)
  {
    value = t;
  }
}
```

Astfel, declaraţiile:

```
int i = 123;
object box = i;
```

corespund conceptual la:

```
int i = 123;
object box = new int_Box(i);
```

Pentru secvenţa:

```
int i = 10;//linia 1
object o = i;//linia 2
int j = (int)o;//linia 3
```

procesul se desfăşoară ca în figura 3.1: la linia 1, se declară şi se iniţializează o variabilă de tip valoare, care va conţine valoarea 10; valoarea va fi stocată pe stivă. La linia 2 se va crea o referinţă o către un obiect alocat în heap, care va conţine atât valoarea 10, cât şi o informaţie despre tipul de dată conţinut (în cazul nostru, System.Int32). Unboxing–ul se face printr–un cast explicit, ca în linia 3.

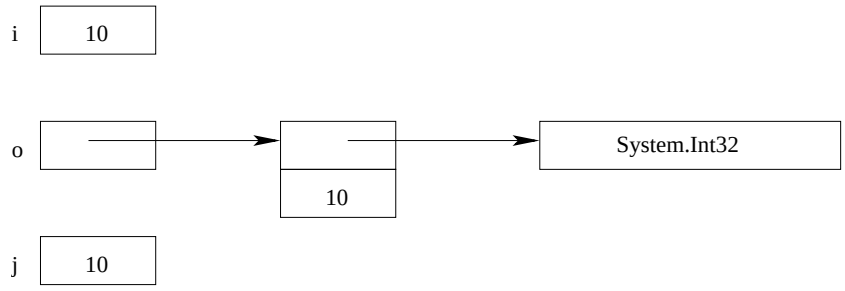

Figura 3.1: Boxing şi unboxing

Determinarea tipului pentru care s–a făcut împachetarea se face prin intermediul operatorului *is*:

```
int i = 123;
object o = i;
if (o is int)
{
  Console.Write("Este un int inauntru!!!!");
}
```

Boxing–ul duce la o clonare a valorii care va fi conţinută. Altfel spus, secvenţa:

```
int i = 10;
object o = i;
i++;
Console.WriteLine("in o: {0}", o);
```

va afişa valoarea nealterată 10.

3.3.5 Declaraţii de variabile şi constante

Variabilele şi constantele trebuie declarate în C#. Opţional, pentru variabile se poate specifica valoarea iniţială, iar pentru constante acest lucru este obligatoriu. O variabilă trebuie să aibă valoarea asignată definită înainte ca valoarea ei să fie utilizată, în cazul în care este declarată în interiorul unei metode. Este o eroare ca într–un sub-bloc să se declare o variabilă cu acelaşi nume ca în blocul conţinător:

```
void F()
{
  int x = 3, y;//ok
  const double d = 1.1;//ok
  {
    string x = "Mesaj: ";//eroare, x mai este declarat
    //in blocul continator
    int z = x + y;//eroare, y nu are o valoare definita asignata
  }
}
```

Constantele au valori iniţiale care trebuie să se poată evalua la compilare.

3.3.6 Declaraţii de etichete

O etichetă poate prefixa o instrucţiune. Ea este vizibilă în întregul bloc şi toate sub-blocurile conţinute. O etichetă poate fi referită de către o instrucţiune *goto*:

```
class DemoLabel
{
  int F(int x)
  {
    if (x >= 0) goto myLabel;
    x = -x;
    myLabel: return x;
  }
  static void Main()
  {
    DemoLabel dl = new DemoLabel();
    dl.f();
  }
}
```

3.3.7 Instrucţiuni de selecţie

Instrucţiunea if

Instrucţiunea if execută o instrucţiune în funcţie de valoarea de adevăr a unei expresii logice. Are formele:

```
if (expresie logica) instructiune;
if (expresie logica) instructiune; else instructiune;
```

Instrucţiunea switch

Permite executarea unei instrucţiuni în funcţie de valoarea unei expresii, care se poate regăsi sau nu într–o listă de valori candidat:

```
switch (expresie)
{
  case eticheta: instructiune;
  case eticheta: instructiune;
  ...
  default: instructiune;
}
```

O etichetă reprezintă o expresie constantă. O instrucţiune poate să şi lipsească şi în acest caz se va executa instrucţiunea de la *case*–ul următor, sau de la *default*. Secţiunea *default* poate să lipsească. Dacă o instrucţiune este nevidă, atunci obligatoriu va avea la sfârşit o instrucţiune *break* sau *goto case expresieConstanta* sau *goto default*.

Expresia după care se face selecţia poate fi de tip *sbyte, byte, short, ushort, int, uint, long, ulong, char, string, enumerare*. Dacă valoarea expresiei se regăseşte printre valorile specificate la clauzele case, atunci instrucţiunea corespunzătoare va fi executată; dacă nu, atunci instrucţiunea de la clauza default va fi executată (dacă ea există). Spre deosebire de C şi C++, e interzis să se folosească fenomenul de "cădere" de la o etichetă la alta; continuarea se face folosind explicit *goto*. Subliniem că se permite folosirea unui selector de tip string.

```
switch (i)
{
  case 0:
         Console.WriteLine("0");
         break;
  case 1:
```

```
        Console.Write("Valoarea ");
        goto case 2;
  case 2:
  case 3:
        Console.WriteLine(i);
        break;
  case 4:
        goto default;
  default:
        Console.WriteLine("Numar in afara domeniului admis");
      break;//neaparat, altfel eroare de compilare
}
```

Remarcăm în exemplul de mai sus că chiar şi în cazul lui *default* e necesar să se folosească instrucţiune de salt (în cazul nostru *break*); o motivaţie ar fi că această clauză *default* nu e necesar să fie trecută ultima în *switch*, ci chiar şi pe prima poziţie – desigur caz mai rar întâlnit.

Există un caz în care *break*, *goto case valoare* sau *goto default* pot să lipsească: când este evident că o asemenea instrucţiune *break/goto* nu ar putea fi atinsă (i.e. sunt prezente instrucţiunile *return*, *throw* sau o ciclare despre care se poate afirma la compilare că este infinită).

3.3.8 Instrucţiuni de ciclare

Există 4 instrucţiuni de ciclare: while, do, for, foreach.

Instrucţiunea while

Permite executarea unei instrucţiuni atâta timp cât valoarea unei expresii logice este adevărată (ciclu cu test anterior).

Sintaxa:

```
while (expresie logica) instructiune;
```

În interiorul unei astfel de instrucţiuni se poate folosi o instrucţiune de salt de tip *break* sau *continue*.

```
while (r != 0)
{
  r = a%b;
  a = b;
  b = a;
}
```

Instrucţiunea do

Execută o instrucţiune o dată sau de mai multe ori, cât timp o condiţie logică este adevărată (ciclu cu test posterior).

Exemplu:

```
do
{
  S += i++;
}while(i<=n)
```

Poate conţine instrucţiuni *break* sau *continue*.

Instrucţiunea for

Execută o secvenţă de iniţializare, după care va executa o instrucţiune atâta timp cât o condiţie este adevărată (ciclu cu test anterior); poate să conţină un pas de reiniţializare (trecerea la pasul următor). Se permite folosirea instrucţiunilor *break* şi *continue*.

Exemplu:

```
for (int i=0; i<n; i++)
{
  Console.WriteLine("i={0}", i);
}
```

Instrucţiunea foreach

Enumeră elementele dintr–o coleţie, executând o instrucţiune pentru fiecare element. Colecţia poate să fie orice instanţă a unei clase care implementează interfaţa *System.Collections.IEnumerable*.

Exemplu:

```
int[] t = {1, 2, 3};
foreach( int x in  t)
{
  Console.WriteLine(x);
}
```

Elementul care se extrage este de tip read–only (deci nu poate fi transmis ca parametru *ref* sau *out* şi nu se poate aplica un operator care să îi schimbe valoarea).

3.3.9 Instrucţiuni de salt

Permit schimbarea ordinii de execuţie a instrucţiunilor. Ele sunt: *break, continue, goto, return, throw.*

Instrucţiunea break

Produce ieşirea forţată dintr–un ciclu de tip *while, do, for, foreach.*

Instrucţiunea continue

Porneşte o nouă iteraţie în interiorul celui mai apropiat ciclu conţinător de tip *while, do, for, foreach.*

Instrucţiunea goto

Goto permite saltul al o anumită instrucţiune. Are 3 forme:

```
goto eticheta;
goto case expresieconstanta;
goto default;
```

Cerinţa este ca eticheta la care se face saltul să fie definită în cadrul funcţiei curente şi saltul să nu se facă în interiorul unor blocuri de instrucţiuni, deoarece nu se poate reface întotdeauna contextul acelui bloc.

Se recomandă evitarea utilizării intense a acestui cuvânt cheie, în caz contrar se poate ajunge la fenomenul de "spagetti code". Pentru o argumentare consistentă a acestei indicaţii, a se vedea articolul clasic al lui Edsger W. Dijkstra, "Go To Statement Considered Harmful": http://www.acm.org/classics/oct95/

Instrucţiunea return

Determină cedarea controlului funţiei apelante de către funcţia apelată. Dacă funcţia apelată are tip de retur, atunci instrucţiunea return trebuie să fie urmată de o expresie care suportă o conversie implicită către tipul de retur.

3.3.10 Instrucţiunile try, throw, catch, finally

Permit tratarea excepţiilor. Vor fi studiate în detaliu la capitolul de excepţii.

3.3.11 Instrucţiunile checked şi unchecked

Controlează contextul de verificare de depăşire a domeniului pentru aritmetica pe întregi şi conversii. Au forma:

```
checked
{
  //instructiuni
}
unchecked
{
  //instructiuni
}
```

Verificare se va face la run–time.

3.3.12 Instrucţiunea lock

Obţine excluderea mutuală asupra unui obiect pentru executarea unui bloc de instrucţiuni. Are forma:

```
lock (x) instructiuni
```

X trebuie să fie de tip referinţă (dacă este de tip valoare, nu se face boxing).

3.3.13 Instrucţiunea using

Determină obţinerea a unei sau mai multor resurse, execută o instrucţiune şi apoi disponibilizează resursa:

```
using ( achizitie de resurse ) instructiune
```

O resursă este o clasă sau o structură care implementează interfaţa *System.IDisposable*, care include o sigură metodă fără parametri *Dispose()*. Achiziţia de resurse se poate face sub formă de variabile locale sau a unor expresii; toate acestea trebuie să fie implicit convertibile la *IDisposable*. Variabilele locale alocate ca resurse sunt read–only. Resursele sunt automat dealocate (prin apelul de *Dispose*) la sfârşitul instrucţiunii (care poate fi bloc de instrucţiuni).

Motivul pentru care există această instrucţiune este unul simplu: uneori se doreşte ca pentru anumite obiecte care deţin resurse importante să se apeleze automat metodă *Dispose()* de dezalocare a lor, cât mai repede cu putinţă.

Exemplu:

```
using System;
using System.IO;
class Test
{
  static void Main()
  {
    using( TextWriter w = File.CreateText("log.txt") )
    {
      w.WriteLine("This is line 1");
      w.EriteLine("This is line 2");
    }
  }
}
```

3.4 Spaţii de nume

În cazul creării de tipuri este posibil să se folosească un acelaşi nume pentru tipurile noi create de către dezvoltatorii de soft. Pentru a putea folosi astfel de clase care au numele comun, dar responsabilităţi diferite, trebuie prevăzută o modalitate de a le adresa în mod unic. Soluţia la această problemă este crearea spaţiilor de nume[4] care rezolvă, printr–o adresare completă astfel de ambiguităţi. Astfel, putem folosi de exemplu clasa Buffer din spaţiul System (calificare completă: System.Buffer), alături de clasa Buffer din spaţiul de nume Curs3: Curs3.Buffer.

Crearea unui spaţiu de nume se face prin folosirea cuvântului namespace:

```
using System;
namespace Curs3
{
  public class Buffer
  {
    public Buffer()
    {
      Console.WriteLine("Bufferul meu!");
    }
  }
}
```

Se pot de asemenea crea spaţii de nume imbricate. Altfel spus, un spaţiu de nume este o colecţie de tipuri sau de alte spaţii de nume.

[4]engl: namespaces

3.4.1 Declaraţii de spaţii de nume

O declaraţie de spaţiu de nume constă în cuvântul cheie *namespace*, urmat de identificatorul spaţiului de nume şi de blocul spaţiului de nume, delimitat de acolade. Spaţiile de nume sunt implicit publice şi acest tip de acces nu se poate modifica. În interiorul unui spaţiu de nume se pot utiliza alte spaţii de nume, pentru a se evita calificarea completă a claselor.

Identificatorul unui spaţiu de nume poate fi simplu sau o secvenţă de identificatori separaţi prin ".". Cea de a doua formă permite definirea de spaţii de nume imbricate, fără a se imbrica efectiv:

```
namespace N1.N2
{
  class A{}
  class B{}
}
```

este echivalentă cu:

```
namespace N1
{
  namespace N2
  {
    class A{}
    class B{}
  }
}
```

Două declaraţii de spaţii de nume cu aceeaşi denumire contribuie la declararea unui acelaşi spaţiu de nume:

```
namespace N1.N2
{
  class A{}
}

namespace N1.N2
{
  class B{}
}
```

este echivalentă cu cele două declaraţii anterioare.

3.4.2 Directiva using

Directiva using facilitează în primul rând utilizarea spaţiilor de nume şi a tipurilor definite în acestea; ele nu creează membri noi în cadrul unităţii de program în care sunt folosite, ci au rol de a uşura referirea tipurilor. Nu se pot utiliza în interiorul claselor, structurilor, enumerăririlor.

Exemplu: e mai uşor de înteles un cod de forma:

```
using System;
class A
{
  static void Main()
  {
    Console.WriteLine("Mesaj");
  }
}
```

decât:

```
class A
{
  static void Main()
  {
    System.Console.WriteLine("Mesaj");
  }
}
```

Directiva using poate fi de fapt folosită atât pentru importuri simbolice, cât şi pentru crearea de aliasuri.

Directiva using pentru import simbolic

O directivă using permite importarea simbolică a tuturor tipurilor conţinute direct într–un spaţiu de nume, i.e. folosirea lor fără a fi necesară o calificare completă. Acest import nu se referă şi la spaţiile de nume conţinute:

```
namespace N1.N2
{
  class A{}
}
namespace N3.N4
{
  class B{};
```

```
}
namespace N5
{
  using N1.N2;
  using N3;
  class C
  {
    A a = null;//ok
    N4.B = null;//Eroare, N4 nu a fost importat
  }
}
```

Importarea de spaţii de nume nu trebuie să ducă la ambiguităţi:

```
namespace N1
{
  class A{}
}
namespace N2
{
  class A{}
}
namespace N3
{
  using N1;
  using N2;
  class B
  {
    A a = null;//ambiguitate: N1.A sau N2.A?
  }
}
```

În situaţia de mai sus, conflictul (care poate foarte uşor în cazul în care se folosesc tipuri produse de dezvoltatori diferiţi) poate fi rezolvat de o calificare completă:

```
namespace N3
{
  using N1;
  using N2;
  class B
  {
```

```
    N1.A a1 = null;
    N2.A a2 = null;//ok, nu mai este ambiguitate
  }
}
```

Tipurile declarate în interiorul unui spaţiu de nume pot avea modificatori de acces *public* sau *internal* (modificatorul implicit). Un tip *internal* nu poate fi folosit prin import în afara assembly-ului, pe când unul *public*, da.

Directiva using ca alias

Introduce un identificator care serveşte drept alias pentru un spaţiu de nume sau pentru un tip.

Exemplu:

```
namespace N1.N2
{
  class A{}
}
namespace N3
{
  using A = N1.N2.A;
  class B
  {
     A a = null;
  }
}
```

Acelaşi efect se obţine creînd un alias la spaţiul de nume N1.N2:

```
namespace N3
{
  using N = N1.N2;
  class B
  {
     N.A a = null;
  }
}
```

Identificatorul dat unui alias trebuie să fie unic, adică în interiorul unui namespace nu trebuie să existe tipuri şi aliasuri cu acelaşi nume:

```
namespace N3
{
  class A{}
}
namespace N3
{
  using A = N1.N2.A;//eroare, deoarece simbolul A mai este definit
}
```

O directivă alias afectează doar blocul în care este definită:

```
namespace N3
{
  using R = N1.N2;
}
namespace N3
{
  class B
  {
    R.A a = null;//eroare, R nu este definit aici
  }
}
```

adică directiva de alias nu este tranzitivă. Situaţia de mai sus se poate rezolva prin declarearea aliasului în afara spaţiului de nume:

```
using R = N1.N2;
namespace N3
{
  class B
  {
    R.A a = null;
  }
}
namespace N3
{
  class C
  {
    R.A b = null;
  }
}
```

Numele create prin directive de alias sunt ascunse de către alte declaraţii care folosesc acelaşi identificator în interiorul unui bloc:

```
using R = N1.N2;
namespace N3
{
  class R{}
  class B
  {
   R.A a;//eroare, clasa R nu are membrul A
  }
}
```

Directivele de alias nu se influenţează reciproc:

```
namespace N1.N2{}
namespace N3
{
  using R1 = N1;
  using R2 = N1.N2;
  using R3 = R1.N2;//eroare, R1 necunoscut
}
```

Curs 4

Clase

4.1 Declararea unei clase

Declararea unei clase se face în felul următor:
atribute$_{opt}$ modificatori-de-clasa$_{opt}$ `class` identificator clasa-de-baza$_{opt}$ corp-clasa ;$_{opt}$

Modificatorii de clasă sunt:

public - clasele publice sunt accesibile de oriunde; poate fi folosit atât pentru clase interioare, cât şi pentru clase care sunt conţinute în spaţii de nume;

internal - se poate folosi atât pentru clase interioare, cât şi pentru clase care sunt conţinute în spaţii de nume (este modificatorul implicit pentru clase care sunt conţinute în spaţii de nume). Semnifică acces permis doar în clasa sau spaţiul de nume care o cuprinde;

protected - se poate specifica doar pentru clase interioare; tipurile astfel calificate sunt accesibile în clasa curentă sau în cele derivate (chiar dacă clasa derivată face parte din alt spaţiu de nume);

private - doar pentru clase interioare; semnifică acces limitat la clasa conţinătoare; este modificatorul implicit;

protected internal - folosibil doar pentru clase interioare; tipul definit este accesibil în spaţiul de nume curent, în clasa conţinătoare sau în tipurile derivate din clasa conţinătoare;

new - permis pentru clasele interioare; clasa astfel calificată ascunde un membru cu acelaşi nume care este moştenit;

sealed - o clasă sealed nu poate fi moştenită; poate fi clasă interioară sau nu;

abstract - clasa care este incomplet definită şi care nu poate fi instanţiată; folosibilă pentru clase interioare sau conţinute în spaţii de nume;

4.2 Membrii unei clase

Corpul unei clase se specifică în felul următor:
{ declaratii-de-membri };$_{opt}$
Membrii unei clase sunt împărţiti în următoarele categorii:

- constante
- câmpuri
- metode
- proprietăţi
- evenimente
- indexatori
- operatori
- constructori (de instanţă)
- destructor
- constructor static
- tipuri

Acestor membri le pot fi ataşaţi modificatorii de acces:

public - membrul este accesibil de oriunde;

protected - membrul este accesabil de către orice membru al clasei conţinătoare şi de către clasele derivate;

internal - membrul este accesabil doar în assembly-ul curent;

protected internal - reuniunea precedentelor două;

private accesabil doar în clasa conţinătoare; este specificatorul implicit.

4.3 Câmpuri

Un câmp reprezintă un membru asociat cu un obiect sau cu o clasă. Modificatorii de câmp care se pot specifica opţional înaintea unui câmp sunt cei de mai sus, la care se adaugă modificatorii *new, readonly, volatile, static*, ce vor fi prezentaţi mai jos. Pentru orice câmp este necesară precizarea unui tip de date, ce trebuie să aibe gradul de accesibilitate cel puţin cu al câmpului ce se declară. Opţional, câmpurile pot fi iniţializate cu valori compatibile. Un astfel de câmp se poate folosi fie prin specificarea numelui său, fie printr-o calificare bazată pe numele clasei sau al unui obiect.

Exemplu:

```
class A
{
  int a;//acces implicit de tip privat
  static void Main()
  {
    A objA = new A();
    objA.a = 1;//se poate accesa in interiorul clasei
  }
}
```

4.3.1 Câmpuri instanţe

Dacă o declaraţie de câmp nu include modificatorul static, atunci acel câmp se va regăsi în orice obiect de tipul clasei curente care va fi instanţiat. Modificări ale acestor câmpuri se vor face independent pentru fiecare obiect. Deoarece un astfel de câmp are o valoare specifică fiecărui obiect, accesarea lui se va face prin calificarea cu numele obiectului:

```
objA.a = 1;
```

(dacă modificatorii de acces permit aşa ceva).

Un câmp special este this (cuvânt cheie) care reprezintă o referinţă la obiectul curent.

4.3.2 Câmpuri statice

Când o declaraţie de câmp include un specificator static, câmpul respectiv nu apaţine fiecărei instanţe în particular, ci clasei însăşi. Accesarea unui câmp static din exteriorul clasei se face exclusiv prin intermediul numelui de clasă:

```
class B
{
  public static int V = 3;
  static void Main()
  {
    B.V++;
  }
}
```

Dacă se face calificarea unui astfel de câmp folosind un nume de obiect se semnalează o eroare de compilare.

4.3.3 Câmpuri readonly

Declararea unui câmp de tip *readonly* (static sau nu) se face prin specificarea cuvântului *readonly* în declaraţia sa:

```
class A
{
  public readonly string salut = ‘‘Salut’’;
  public readonly string nume;
  public class A(string nume)
  {
    this.nume = nume;
  }
}
```

Atribuirea asupra unui câmp de tip *readonly* se poate face doar la declararea sa (ca în exemplu de mai sus) sau prin intermediul unui constructor. Valoarea unor astfel de câmpuri nu se presupune a fi cunoscută la compilare.

4.3.4 Câmpuri volatile

Modificatorul "volatile" se poate specifica doar pentru tipurile:

- byte, sbyte, short, ushort, int, uint, char, float, bool;
- un tip enumerare având tipul de reprezentare byte, sbyte, short, ushort, int, uint;
- un tip referinţă

Pentru câmpuri nevolatile, tehnicile de optimizare care reordonează instrucţiunile pot duce la rezultate neaşteptate sau nepredictibile în programe multithreading care accesează câmpurile fără sincronizare (efectuabilă cu instrucţiunea lock). Aceste optimizări pot fi făcute de către compilator, de către sistemul de rulare[1] sau de către hardware. Următoarele tipuri de optimizări sunt afectate în prezenţa unui modificator volatile:

- citirea unui câmp *volatile* este garantată că se va întâmpla înainte de orice referire la câmp care apare după citire;
- orice scriere a unui câmp *volatile* este garantată că se va petrece după orice instrucţiune anterioară care se referă la câmpul respectiv.

4.3.5 Iniţializarea câmpurilor

Pentru fiecare câmp declarat se va asigna o valoare implicită astfel:

- numeric: 0
- bool: false
- char: '\0'
- enum: 0
- referinţă: null

4.4 Constante

O constantă este un câmp a cărui valoare poate fi calculată la compilare. O constantă poate fi prefixată de următorii modificatori: *new, public, protected, internal, protected internal,private*. Cuvantul *new* poate să se combine cu unul din ceilalţi 4 modificatori de acces. Pentru un câmp constant e obligatoriu să se asigneze o valoare calculabilă la compilare:

```
class A
{
  public const int n=2;
}
```

[1] Engl: runtime system

Tipul unei constante poate fi *sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double decimal, bool, string, enum*, referinţă. Valoarea care se asigneză unei constante trebuie să admită o conversie implicită către tipul constantei. Tipul unei constante trebuie să fie cel puţin la fel de accesibil ca şi constanta însăşi.

Orice câmp constant este automat un câmp static. Un câmp constant diferă de un câmp static *readonly*: *const*-ul are o valoare cunoscută la compilare, pe când valoarea unui *readonly* poate fi iniţializată la runtime în interiorul constructorului (cel mai târziu, de altfel).

4.5 Metode

O metodă este un membru care implementează un calcul sau o acţiune care poate fi efectuată de către un obiect sau o clasă. Antetul unei metode se declară în felul următor:
atribute$_{opt}$ modificator-de-metoda$_{opt}$ tip-de-retur nume (lista-parametrilor-formali$_{opt}$) corp-metoda
unde modificator-de-metoda poate fi:

- orice modificator de acces
- new
- static
- virtual
- sealed
- override
- abstract
- extern

Tipul de retur poate fi orice tip de dată care este cel puţin la fel de accesibil ca şi metoda însăşi sau void (absenţa informaţiei returnate); nume poate fi un identificator de metodă din clasa curentă sau un identificator calificat cu numele unei interfeţe pe care o implementează (*NumeInterfata.numeMetoda*); parametrii pot fi de tip *ref, out, params*, sau fără nici un calificator; corp-metoda este un bloc cuprins între acolade sau doar caracterul ";" (dacă este vorba de o metodă ce nu se implementează în tipul curent).

Despre calificatorii *virtual, override, sealed, new, abstract* se va discuta mai pe larg într–o secţiune viitoare.

4.5.1 Metode statice şi nestatice

O metodă se declară a fi statică dacă numele ei este prefixat cu modificatorul de metodă static. O astfel de metodă nu operează asupra unei instanţe anume, ci doar asupra clasei. Este o eroare ca o metodă statică să facă referire la un membru nestatic al unei clase. Apelul unei astfel de metode se face prin *NumeClasa.NumeMetoda* sau direct *NumeMetoda* dacă este apelată din context static al aceleiaşi clase.

O metodă nestatică nu are cuvântul "static" specificat; ea este apelabilă în conjuncţie cu un obiect anume.

4.5.2 Metode externe

Metodele externe se declară folosind modificatorul extern; acest tip de metode sunt implementate extern, de obicei în alt limbaj decât C#. Deoarece o astfel de metodă nu conţine o implementare, corpul acestei metode este ";".

Exemplu: se utilizează metoda MessageBox importată din biblioteca dll User32.dll:

```
using System;
using System.Runtime.InteropServices;
class Class1
{
  [DllImport("User32.dll")]
  public static extern int MessageBox(int h, string m, string c, int type);
  static void Main(string[] args)
  {
    int retVal = MessageBox(0, "Hello", "Caption", 0);
  }
}
```

4.6 Proprietăţi

O proprietate este un membru care permite acces la un atribut al unui obiect sau al unei clase. Exemple de proprietăţi sunt: lungimea unui şir, numele unui client, textul continut într–un control de tip TextBox. Proprietăţile sunt extensii naturale ale câmpurilor, cu deosebirea că ele nu presupun alocarea de memorie. Ele sunt de fapt nişte metode (accesori) care permit citirea sau setarea unor atribute ale unui obiect sau clase; reprezintă modalitatea de scriere a unor metode de tip get/set pentru clase sau obiecte.

Declararea unei proprietăţi se face astfel:
atribute$_{opt}$ modificatori-de-proprietate$_{opt}$ tip nume {declaratii-de-accesor}
Modificatorii de proprietate sunt: *new, public, protected, internal, private, static, virtual, sealed, override, abstract, extern.*

Tipul unei proprietăţi specifică tipul de proprietate introdus de declaraţie, i.e. ce valori vor putea fi atribuite proprietăţii respective (dacă accesorul de tip *set* a fost definit), respectiv care este tipul valorii returnate de această proprietate (corespunzător accesorului de tip *get*).

Exemplu:

```
using System;
class Circle
{
  private double radius;
  public double Radius
  {
    get
    {
      return radius;
    }
    set
    {
      radius = value;
    }
  }
  public double Area
  {
    get
    {
      return Math.PI * radius * radius;
    }
    set
    {
      radius = Math.Sqrt(value/Math.PI);
    }
  }
}

class Test
{
  static void Main()
```

```
  {
    Circle c = new Circle();
    c.Radius = 10;
    Console.WriteLine(‘‘Area: {0}’’, c.Area);
    c.Area = 15;
    Console.WriteLine(‘‘Radius: {0}’’. c.Radius)
  }
}
```

Un accesor de tip *get* corespunde unei metode fără parametri, care returnează o valoare de tipul proprietăţii. Când o proprietate este folosită într–o expresie, accesorul *get* este o apelat pentru a returna valoarea cerută.

Un accesor de tip *set* corespunde unei metode cu un singur parametru de tipul proprietăţii şi tip de retur *void*. Acest parametru implicit al lui *set* este numit întotdeauna *value*. Când o proprietate este folosită ca destinatar într–o atribuire, sau când se folosesc operatorii ++ şi −−, accesorului *set* i se transmite un parametru care reprezintă noua valoare.

În funcţie de prezenţa sau absenţa accesorilor, o proprietate este clasificată după cum urmează:

proprietate read–write, dacă are ambele tipuri de accesori;

proprietate read–only, dacă are doar accesor de tip *get*; este o eroare de compilare să se facă referire în program la o proprietate în sensul în care s–ar cere operarea cu un accesor de tip *set*;

proprietate write–only, dacă este prezent doar accesorul de tip *set*; este o eroare de compilare utilizarea unei proprietăţi într–un context în care ar fi necesară prezenţa accesorului *get*.

Demn de menţionat este că ei pot fi folosiţi nu numai pentru a asigura o sintaxă simplu de folosit pentru metodele tradiţionale de tip get/set, ci şi pentru scrierea controalelor .NET utilizator, care se face extrem de uşor.

În figura 4.1 este dată reprezentarea unui control utilizator:

Codul corespunzător este dat mai jos:

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Drawing;
using System.Data;
using System.Windows.Forms;
```

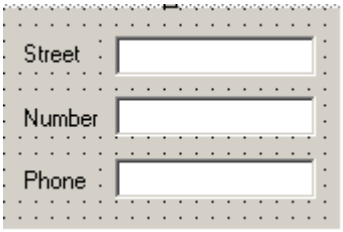

Figura 4.1: Control definit de utilizator

```
namespace UserControlSample
{
  public class UserControl1 : System.Windows.Forms.UserControl
  {
    private System.Windows.Forms.Label label1;
    private System.Windows.Forms.TextBox streetTextBox;
    private System.Windows.Forms.Label label2;
    private System.Windows.Forms.TextBox numberTextBox;
    private System.Windows.Forms.Label label3;
    private System.Windows.Forms.TextBox phoneTextBox;
    private System.ComponentModel.Container components = null;
    public UserControl1()
    {
      // This call is required by the Windows.Forms Form Designer.
      InitializeComponent();
    }

    protected override void Dispose( bool disposing )
    {
      if( disposing )
      {
         if( components != null )
           components.Dispose();
      }
      base.Dispose( disposing );
    }
    #region Component Designer generated code

    private void InitializeComponent()
    {
      this.label1 = new System.Windows.Forms.Label();
```

```
this.streetTextBox = new System.Windows.Forms.TextBox();
this.label2 = new System.Windows.Forms.Label();
this.numberTextBox = new System.Windows.Forms.TextBox();
this.label3 = new System.Windows.Forms.Label();
this.phoneTextBox = new System.Windows.Forms.TextBox();
this.SuspendLayout();
this.label1.AutoSize = true;
this.label1.Location = new System.Drawing.Point(8, 16);
this.label1.Name = "label1";
this.label1.Size = new System.Drawing.Size(34, 13);
this.label1.TabIndex = 0;
this.label1.Text = "Street";
this.streetTextBox.Location = new System.Drawing.Point(56, 14);
this.streetTextBox.Name = "streetTextBox";
this.streetTextBox.TabIndex = 1;
this.streetTextBox.Text = "";
this.label2.AutoSize = true;
this.label2.Location = new System.Drawing.Point(8, 48);
this.label2.Name = "label2";
this.label2.Size = new System.Drawing.Size(44, 13);
this.label2.TabIndex = 2;
this.label2.Text = "Number";
this.numberTextBox.Location = new System.Drawing.Point(56, 44);
this.numberTextBox.Name = "numberTextBox";
this.numberTextBox.TabIndex = 3;
this.numberTextBox.Text = "";
this.label3.AutoSize = true;
this.label3.Location = new System.Drawing.Point(8, 79);
this.label3.Name = "label3";
this.label3.Size = new System.Drawing.Size(37, 13);
this.label3.TabIndex = 4;
this.label3.Text = "Phone";
this.phoneTextBox.Location = new System.Drawing.Point(56, 75);
this.phoneTextBox.Name = "phoneTextBox";
this.phoneTextBox.TabIndex = 5;
this.phoneTextBox.Text = "";
this.Controls.AddRange(new System.Windows.Forms.Control[] {
this.phoneTextBox,
this.label3,
this.numberTextBox,
this.label2,
```

```
      this.streetTextBox,
      this.label1});
      this.Name = "UserControl1";
      this.Size = new System.Drawing.Size(168, 112);
      this.ResumeLayout(false);
    }
    #endregion

    [Category ("Data"), Description ("Contents of Street Control")]
    public string Street
    {
      get{ return streetTextBox.Text; }
      set{ streetTextBox.Text = value; }
    }

    [Category ("Data"), Description ("Contents of Number Control")]
    public string Number
    {
      get{ return numberTextBox.Text; }
      set{ numberTextBox.Text = value; }
    }

    [Category ("Data"), Description ("Contents of Phone Control")]
    public string Phone
    {
      get{ return phoneTextBox.Text; }
      set{ phoneTextBox.Text = value; }
    }
  }
}
```

Interesante sunt aici proprietăţile publice Street, Number şi Phone care vor fi vizibile în fereastra *Properties* atunci când acest control va fi adăugat la o formă. Atributele cuprinse între paranteze drepte sunt opţionale, dar vor face ca aceste proprietăţi să fie grupate în secţiunea de date a ferestrei *Properties*, şi nu în cea "Misc". Pentru mai multe detalii despre componente utilizator, a se vedea [1], cap. 4.

4.7 Indexatori

Uneori are sens tratarea unei clase ca fiind un array. Este o generalizare a supraîncărcării operatorului [] din C++, fiind o facilitate ce dă o mare flexibilitate.

Declararea unui indexator se face în felul următor:
atribute$_{opt}$ modificatori-de-indexator$_{opt}$ declarator-de-indexator {declaratii-de-accesor}

Modificatorii de indexator pot fi: *new, public, protected, internal, private, virtual, sealed, override, abstract, extern.* Declaratorul de indexator are forma:
tip-de-retur this[lista-parametrilor-formali].
Lista parametrilor formali trebuie să conţină cel puţin un parametru şi nu poate să aibe vreun parametru de tip *ref* sau *out.* Declaraţiile de accesor vor conţine accesor get sau accesor set, asemănător cu cei de la proprietăţi.

Exemple:

1. Exemplul 1: un indexator simplu:

```
using System;
class MyVector
{
  private double[] v;
  public MyVector( int length )
  {
    v = new double[ length  ];
  }
  public int Length
  {
    get
    {
      return length;
    }
  }
  public double this[int index]
  {
    get
    {
      return v[ index];
    }
    set
```

```
    {
      v[index] = value;
    }
  }
}

class Test
{
  static void Main()
  {
    MyVector v = new MyVector( 10 );
    v[0] = 0;
    v[1] = 1;
    for( int i=2; i<v.Length; i++)
    {
      v[i] = v[i-1] + v[i-2];
    }
    for( int i=0; i<v.Length; i++)
    {
      Console.WriteLine(``v[`` + i.ToString() + ``]='' + v[i]);
    }
  }
}
```

2. Exemplul 2: supraîncărcarea indexatorilor:

```
using System;
using System.Collections;
class DataValue
{
  public DataValue(string name, object data)
  {
    this.name = name;
    this.data = data;
  }
  public string Name
  {
    get
    {
      return(name);
    }
```

```
    set
    {
      name = value;
    }
  }

  public object Data
  {
    get
    {
      return(data);
    }
    set
    {
      data = value;
    }
  }
  string name;
  object data;
}

class DataRow
{
  ArrayList row;
  public DataRow()
  {
    row = new ArrayList();
  }

  public void Load()
  {
    row.Add(new DataValue("Id", 5551212));
    row.Add(new DataValue("Name", "Fred"));
    row.Add(new DataValue("Salary", 2355.23m));
  }

  public object this[int column]
  {
    get
    {
      return(row[column - 1]);
```

```
    }
    set
    {
      row[column - 1] = value;
    }
  }

  int FindColumn(string name)
  {
    for (int index = 0; index < row.Count; index++)
    {
      DataValue dataValue = (DataValue) row[index];
      if (dataValue.Name == name)
      return(index);
    }
    return(-1);
  }

  public object this[string name]
  {
    get
    {
      return this[FindColumn(name)];
    }
    set
    {
      this[FindColumn(name)] = value;
    }
  }
}

class Test
{
  public static void Main()
  {
    DataRow row = new DataRow();
    row.Load();
    DataValue val = (DataValue) row[0];
    Console.WriteLine("Column 0: {0}", val.Data);
    val.Data = 12; // set the ID
    DataValue val = (DataValue) row["Id"];
```

```
    Console.WriteLine("Id: {0}", val.Data);
    Console.WriteLine("Salary: {0}",
    ((DataValue) row["Salary"]).Data);
    ((DataValue)row["Name"]).Data = "Barney"; // set the name
    Console.WriteLine("Name: {0}", ((DataValue) row["Name"]).Data);
  }
}
```

3. Exemplul 3: Indexator cu mai mulţi parametri:

```
using System;
namespace MyMatrix
{
  class Matrix
  {
    double[,] matrix;

    public Matrix( int rows, int cols )
    {
      matrix = new double[ rows, cols];
    }

    public double this[int i, int j]
    {
      get
      {
        return matrix[i,j];
      }
      set
      {
        matrix[i,j] = value;
      }
    }

    public int RowsNo
    {
      get
      {
        return matrix.GetLength(0);
      }
    }
```

```
        public int ColsNo
        {
          get
          {
            return matrix.GetLength(1);
          }
        }

        static void Main(string[] args)
        {
          MyMatrix m = new MyMatrix(2, 3);
          Console.WriteLine("Lines: {0}", m.RowsNo);
          Console.WriteLine("Columns: {0}", m.ColsNo);
          for(int i=0; i<m.RowsNo; i++)
          {
            for( int j=0; j<m.ColsNo; j++)
            {
              m[i,j] = i + j;
            }
          }
          for(int i=0; i<c.RowsNo; i++)
          {
            for( int j=0; j<c.ColsNo; j++)
            {
              Console.Write(c[i,j] + " ");
            }
            Console.WriteLine();
          }
        }
      }
    }
```

4.8 Operatori

Un operator este un membru care defineşte semnificaţia unei expresii operator care poate fi aplicată unei instanţe a unei clase. Corespunde supraîncărcării operatorilor din C++. O declaraţie de operator are forma:
atribute$_{opt}$ modificatori-de-operator declaratie-de-operator corp-operator
Se pot declara operatori unari, binari şi de conversie.

Următoarele reguli trebuie să fie respectate pentru orice operator:

1. Orice operator trebuie să fie declarat public şi static.

2. Parametrii unui operator trebuie să fie de tip valoare; nu se admite să fie de tip ref sau out.

3. Acelaşi modificator nu poate apărea de mai multe ori în antetul unui operator

4.8.1 Operatori unari

Supraîncărcarea operatorilor unari are forma:
tip operator operator-unar-supraincarcabil (tip identificator) corp
Operatorii unari supraîncărcabili sunt: + - ! ˜ ++ – true false. Următoarele reguli trebuie să fie respectate la supraîncărcarea unui operator unar (T reprezintă clasa care conţine definiţia operatorului):

1. Un operator +, -, !, ˜ trebuie să preia un singur parametru de tip T şi poate returna orice tip.

2. Un operator ++ sau – trebuie să preia un singur parametru de tip T şi trebuie să returneze un rezultat de tip T.

3. Un operator unar true sau false trebuie să preia un singur parametru de tip T şi să returneze bool.

Operatorii true şi false trebuie să fie ori ambii definiţi, ori nici unul (altfel apare o eroare de compilare). Ei sunt necesari pentru cazuri de genul:

```
if( a==true )
```

sau

```
if( a==false )
```

Mai general, ei pot fi folosiţi ca expresii de control în if, do, while şi for, precum şi în operatorul ternar "? :". Deşi pare paradoxal, nu este obligatoriu ca `if (a==true)` să fie echivalentă cu `if (!(a==false))`, de exemplu pentru tipuri SQL care pot fi null, ceea ce nu înseamă nici true, nici false, ci lipsă de informaţie.

Exemplu:

```
public struct DBBool
{
  private int x;
  public static bool operator true(DBBool x)
  {
    return x.value > 0;
  }
  public static bool operator false(DBBool x)
  {
    return x.value <= 0;
  }
  ...
}
```

Exemplul de mai jos arată modul în care se face supraîncărcarea operatorului ++, care poate fi folosit atât ca operator de preincrementare cât şi ca operator de postincrementare:

```
public class IntVector
{
  public int Length { ... } // read-only property
  public int this[int index] { ... } // read-write indexer
  public IntVector(int vectorLength) { ... }
  public static IntVector operator++(IntVector iv)
  {
    IntVector temp = new IntVector(iv.Length);
    for (int i = 0; i < iv.Length; ++i)
      temp[i] = iv[i] + 1;
    return temp;
  }
}

class Test
{
  static void Main()
  {
    IntVector iv1 = new IntVector(4); // vector of 4x0
    IntVector iv2;
    iv2 = iv1++; // iv2 contains 4x0, iv1 contains 4x1
    iv2 = ++iv1; // iv2 contains 4x2, iv1 contains 4x2
  }
}
```

4.8.2 Operatori binari

Declararea unui operator binar se face astfel:
tip operator operator-binar-supraincarcabil (tip identificator, tip identificator) corp
Operatorii binari supraîncărcabili sunt: `+ - * / % & | ^ << >> == != > < >= <=`.
Cel puţin unul dintre cei doi parametri preluaţi trebuie să fie de tipul conţinător.
Operatorii de shiftare trebuie să aibă primul parametru de tipul clasei în care se declară, iar al doilea parametru de tip int. Unii operatori trebuie să se declare în pereche:

1. operatorii `==` şi `!=`
2. operatorii `>` şi `<`
3. operatorii `>=` şi `<=`

Pentru operaratorul `==`, este indicată şi definirea metodei Equals(), deoarece tipul respectiv va putea fi astfel folosit şi de către limbaje care nu suportă supraîncărcarea operatorilor, dar pot apela metoda polimorfică Equals() definită în clasa object.

Nu se pot supaîncărca operatorii $+=, -=, /=, *=$; dar pentru ca aceştia să funcţioneze, este suficient să se supraîncarce operatorii corespunzători: $+, -, /, *$.

4.8.3 Operatori de conversie

O declaraţie de operator de conversie trebuie introduce o conversie definită de utilizator, care se va adăuga (dar nu va suprascrie) la conversiile predefinite. Declararea unui operator de conversie se face astfel:
implicit operator tip (tip parametru) corp
explicit operator tip (tip parametru) corp
După cum se poate deduce, conversiile pot fi implicite sau explicite. Un astfel de operator va face conversia de la un tip sursă, indicat de tipul parametrului din antet la un tip destinaţie, indicat de tipul de retur. O clasă poate să declare un operator de conversie de la un tip sursă S la un tip destinaţie T cu următoarele condiţii:

1. S şi T sunt tipuri diferite
2. Unul din cele două tipuri este clasa în care se face definirea.
3. T şi S nu sunt object sau tip interfaţă.

4. T şi S nu sunt baze una pentru cealaltă.

Un bun design asupra operatorilor de conversie are în vedere următoarele:

- Conversiile implicite nu ar trebui să ducă la pierdere de informaţie sau la apariţia de excepţii;
- Dacă prima condiţie nu este îndeplinită, atunci neapărat trebuie declarată ca o conversie explicită.

Exemplu:

```
using System;
public class Digit
{
  byte value;
  public Digit(byte value)
  {
    if (value < 0 || value > 9) throw new ArgumentException();
    this.value = value;
  }
  public static implicit operator byte(Digit d)
  {
    return d.value;
  }
  public static explicit operator Digit(byte b)
  {
    return new Digit(b);
  }
}
```

Prima conversie este implicită pentru că nu va duce la pierderea de informaţie. Cea de doua poate să arunce o excepţie (via constructor) şi de aceea este declarată ca şi conversie explicită.

4.8.4 Exemplu: clasa Fraction

```
using System;
public class Fraction
{
  public Fraction(int numerator, int denominator)
  {
    Console.WriteLine("In constructor Fraction(int, int)");
```

```
    this.numerator=numerator;
    this.denominator=denominator;
  }
  public Fraction(int wholeNumber)
  {
    Console.WriteLine("In Constructor Fraction(int)");
    numerator = wholeNumber;
    denominator = 1;
  }

  public static implicit operator Fraction(int theInt)
  {
    System.Console.WriteLine("In conversie implicita la Fraction");
    return new Fraction(theInt);
  }
  public static explicit operator int(Fraction theFraction)
  {
    System.Console.WriteLine("In conversie explicita la int");
    return theFraction.numerator /
    theFraction.denominator;
  }
  public static bool operator==(Fraction lhs, Fraction rhs)
  {
    Console.WriteLine("In operator ==");
    if (lhs.denominator * rhs.numerator ==
      rhs.denominator * lhs.numerator )
    {
      return true;
    }
    return false;
  }
  public static bool operator !=(Fraction lhs, Fraction rhs)
  {
    Console.WriteLine("In operator !=");
    return !(lhs==rhs);
  }
  public override bool Equals(object o)
  {
    Console.WriteLine("In metoda Equals");
    if (! (o is Fraction) )
    {
```

```
      return false;
    }
    return this == (Fraction) o;
  }
  public static Fraction operator+(Fraction lhs, Fraction rhs)
  {
    Console.WriteLine("In operator+");
    // 1/2 + 3/4 == (1*4) + (3*2) / (2*4) == 10/8
    int firstProduct = lhs.numerator * rhs.denominator;
    int secondProduct = rhs.numerator * lhs.denominator;
    return new Fraction(
    firstProduct + secondProduct,
    lhs.denominator * rhs.denominator
    );
    //ar mai trebui facuta reducerea termenilor
  }
  public override string ToString( )
  {
    String s = numerator.ToString( ) + "/" +
    denominator.ToString( );
    return s;
  }
  private int numerator;
  private int denominator;
}
public class Tester
{
  static void Main( )
  {
    Fraction f1 = new Fraction(3,4);
    Console.WriteLine("f1: {0}", f1.ToString( ));
    Fraction f2 = new Fraction(2,4);
    Console.WriteLine("f2: {0}", f2.ToString( ));
    Fraction f3 = f1 + f2;
    Console.WriteLine("f1 + f2 = f3: {0}", f3.ToString( ));
    Fraction f4 = f3 + 5;
    Console.WriteLine("f3 + 5 = f4: {0}", f4.ToString( ));
    Fraction f5 = new Fraction(2,4);
    if (f5 == f2)
    {
      Console.WriteLine("F5: {0} == F2: {1}",
```

```
      f5.ToString( ),
      f2.ToString( ));
    }
  }
}
```

4.9 Constructori de instanţă

Un constructor de instanţă este un membru care implementează acţiuni care sunt cerute pentru a iniţializa o instanţă a unei clase. Declararea unui astfel de constructor se face în felul următor:
atribute$_{opt}$ modificatori-de-constructor declarator-de-constructor corp-constructor
Un modificator de constructor poate fi: *public, protected, internal, private, extern.* Un declarator de constructor are forma:
nume-clasa (lista-parametrilor-formali$_{opt}$) initializator-de-constructor$_{opt}$
unde initializatorul-de-constructor are forma:
: base(lista-argumente$_{opt}$) sau
: this(lista-argumente$_{opt}$).
Corp-constructor poate fi: un bloc de declaraţii şi instrucţiuni delimitat de acolade sau caracterul punct şi virgulă.

Un constructor are acelaşi nume ca şi clasa din care face parte şi nu returnează un tip. Constructorii de instanţă nu se moştenesc. Dacă o clasă nu conţine nici o declaraţie de constructor de instanţă, atunci compilatorul va crea automat unul implicit. O clasă care este moştenită dintr-o altă clasă ce nu are constructori fără parametri va trebui să utilizeze un apel de constructor de clasă de bază pentru care să furnizeze parametrii potriviţi; acest apel se face prin intermediul iniţializatorului de constructor. Un constructor poate apela la un alt constructor al clasei din care face parte pentru a efectua iniţializări. Când există câmpuri instanţă care au o expresie de iniţializare în afara constructorilor clasei respective, atunci aceste iniţializări se vor face înainte de apelul de constructor al clasei de bază. Pentru a nu se permite crearea sau moştenirea unei clase trebuie ca această clasă să conţină cel puţin un constructor definit de utilizator şi toţi constructorii să fie declaraţi privaţi.

4.10 Constructori statici

Un constructor static este un membru care implementează acţiunile cerute pentru iniţializara unei clase. Declararea unui constructor static se face ca mai jos:

atribute$_{opt}$ modificator-de-constructor-static identificator( ) corp
Modificatorii de constructori statici se pot da sub forma:
extern$_{opt}$ static sau
static extern$_{opt}$
Constructorii statici nu se moştenesc, nu se pot apela direct şi nu se pot supraîncărca. Un constructor static se va executa cel mult o dată într-o aplicaţie. Se garantează faptul că acest constructor se va apela înaintea creării vreunei instanţe a clasei respective sau înaintea accesului la un membru static. Acest apel este nedeterminist, necunoscându–se exact când sau dacă se va apela. Un astfel de constuctor nu are specificator de acces şi poate să acceseze doar membri statici.

Exemplu:

```
class Color
{
  public Color(int red, int green, int blue)
  {
    this.red = red;
    this.green = green;
    this.blue = blue;
  }
  int red;
  int green;
  int blue;

  public static readonly Color Red;
  public static readonly Color Green;
  public static readonly Color Blue;
  // constructor static
  static Color()
  {
    Red = new Color(255, 0, 0);
    Green = new Color(0, 255, 0);
    Blue = new Color(0, 0, 255);
  }
}
class Test
{
  static void Main()
  {
    Color background = Color.Red;
```

```
  }
}
```

4.11 Clase interioare

O clasă conţine membri, iar în particular aceştia pot fi şi clase.
Exemplul 1:

```
using System;
class A
{
  class B
  {
    public static void F()
    {
      Console.WriteLine(‘‘A.B.F’’);
    }
  }
  static void Main()
  {
    A.B.F();
  }
}
```

Exemplul 2:

```
public class List
{
  // Private data structure
  private class Node
  {
    public object Data;
    public Node Next;
    public Node(object data, Node next)
    {
      this.Data = data;
      this.Next = next;
    }
  }
  private Node first = null;
  private Node last = null;
```

```
  //Interfata publica
  public void AddToFront(object o) {...}
  public void AddToBack(object o) {...}
  public object RemoveFromFront() {...}
  public object RemoveFromBack() {...}
  public int Count { get {...} }
}
```

Accesarea unei clase interioare se face prin *NumeClasaExterioara.NumeClasaInterioara* (aşa cum este arătat în Exemplul 1), de unde se deduce că o clasă interioară se comportă ca un membru static al tipului conţinător. O clasă declarată în interiorul unei alte clase poate avea unul din gradele de accesibilitate *public, protected internal, protected, internal, private* (implicit este *private*). O clasă declarată în interiorul unei structuri poate fi declarată *public, internal* sau *private* (implicit *private*).

Crearea unei instanţe a unei clase interioare nu trebuie să fie precedată de crearea unei instanţe a clasei exterioare conţinătoare, aşa cum se vede din Exemplul 1. O clasă interioară nu au vreo relaţie specială cu membrul predefinit *this* al clasei conţinătoare. Altfel spus, nu se poate folosi *this* în interiorul unei clase interioare pentru a accesa membri instanţă din tipul conţinător. Dacă o clasă interioară are nevoie să acceseze membri instanţă ai clasei conţinătoare, va trebui să primească prin constructor parametrul *this* care să se referă la o astfel de instanţă:

```
using System;
class C
{
  int i = 123;
  public void F()
  {
    Nested n = new Nested(this);
    n.G();
  }

  public class Nested
  {
    C this_c;
    public Nested(C c)
    {
      this_c = c;
    }
    public void G()
```

```
    {
      Console.WriteLine(this_c.i);
    }
  }
}
class Test
{
  static void Main()
  {
    C c = new C();
    c.F();
  }
}
```

Se observă de mai sus că o clasă interioară poate manipula toţi membrii care sunt accesibili în interiorul clasei conţinătoare, indiferent de gradul lor de accesibilitate. În cazul în care clasa exterioară (conţinătoare) are membri statici, aceştia pot fi utilizaţi fără a se folosi numele clasei conţinătoare:

```
using System;
class C
{
  private static void F()
  {
    Console.WriteLine("C.F");
  }
  public class Nested
  {
    public static void G()
    {
      F();
    }
  }
}
class Test
{
  static void Main()
  {
    C.Nested.G();
  }
}
```

Clasele interioare se folosesc intens în cadrul containerilor pentru care trebuie să se construiască un enumerator. Clasa interioară va fi în acest caz strâns legată de container şi va duce la o implementare uşor de urmărit şi de întreţinut.

Curs 5

Destructori. POO în C#

5.1 Destructori

Managementul memoriei este făcut sub platforma .NET în mod automat, de către garbage collector, parte componentă a CLR–ului. Pentru o explicaţie asupra modului cum acţionează garbage collector-ul, a se vedea [5], pag. 225 şi urm.

În general, acest garbage collector scuteşte programatorul de grija dealocării memoriei. Dar în anumite situaţii, se doreşte să se facă management manual al dealocării resurselor (de exemplu al resurselor care ţin de sistemul de operare: fişiere, conexiuni la reţea sau la baza de date, ferestre, etc, sau al altor resurse al căror management nu se face de către CLR). În C# există posibilitatea de a lucra cu destructori, sau cu metode de tipul *Dispose(), Close()*.

Un destructor se declară în felul următor:
atribute$_{opt}$ extern$_{opt}$ ~identificator() corp-destructor
unde identificator este numele clasei. Un destructor nu are modificator de acces, nu poate fi apelat manual, nu poate fi supraîncărcat, nu este moştenit.

Un destructor este o scurtătură sintactică pentru metoda *Finalize()*, care este definită în clasa *System.Object*. Programatorul nu poate să suprascrie sau să apeleze această metodă.

Exemplu:

```
~MyClass()
{
   // Perform some cleanup operations here.
}
```

Metoda de mai sus este automat translatată în:

```
protected override void Finalize()
{
   try
   {
      // Perform some cleanup operations here.
   }
   finally
   {
      base.Finalize();
   }
}
```

Problema cu destructorul este că el e chemat doar de către garbage collector, dar acest lucru se face nedeterminist (cu toate că apelarea de destructor se face în cele din urmă, dacă programatorul nu împiedică explicit acest lucru).

Există cazuri în care programatorul doreşte să facă dealocarea manual, astfel încât să nu aştepte ca garbage collectorul să apeleze destructorul. Programatorul poate scrie o metoda care să facă acest lucru. Se sugerează definirea unei metode *Dispose()* care ar trebui să fie explicit apelată atunci când resurse de sistem de operare trebuie să fie eliberate. În plus, clasa respectivă ar trebui să implementeze interfaţa *System.IDisposable*, care conţine această metodă.

În acest caz, *Dispose()* ar trebui să inhibe executarea ulterioară de garbage collector pentru instanţa curentă. Această manevră permite evitarea eliberării unei resurse de două ori. Dacă clientul nu apelează explicit *Dispose()*, atunci garbage collectorul va apela el destructorul la un moment dat. Întrucât utilizatorul poate să nu apeleze *Dispose()*, este necesar ca tipurile care implemetează această metodă să definească de asemenea destructor.

Exemplu:

```
public class ResourceUser: IDisposable
{
  public void Dispose()
  {
    hwnd.Release();//elibereaza o fereastra in Win32
    GC.SuppressFinalization(this);//elimina apel de Finalize()
  }

  ~ResourceUser()
  {
    hwnd.Release();
```

```
  }
}
```

Pentru anumite clase C# se pune la dispoziţie o metodă numită *Close()* în locul uneia *Dispose()*: fisiere, socket-uri, ferestre de dialog, etc. Este indicat ca să se adauge o metodă *Close()* care să facă doar apel de *Dispose()*:

```
//in interiorul unei clase
public void Close()
{
  Dispose();
}
```

Modalitatea cea mai indicată este folosirea unui bloc *using*, caz în care se va elibera obiectul alocat (via metoda Dispose()) la sfârşitul blocului:

```
using( obiect )
{
  //cod
}//aici se va apela automat metoda Dispose()
```

5.2 Specializarea şi generalizarea

Specializarea reprezintă o tehnică de a obţine noi clase pornind de la cele existente. Deseori între clasele pe care le modelăm putem observa relaţii de genul "este un/o": un om este un mamifer, un salariat este un angajat, etc. Toate acestea duc la crearea unei ierarhii de clase, în care din clase de bază (mamifer sau angajat) descind alte clase, care pe lângă câmpuri din clasa de bază mai au şi caracteristici proprii. Obţinerea unei clase derivate plecând de la altă clasă se numeşte specializare, operaţia inversă purtând numele de generalizare. O clasă de bază defineşte un tip comun, compatibil cu oricare din clasele derivate (direct sau indirect).

În C# o clasă nu trebuie să moştenească explicit din altă clasă; în acest caz se va considera că ea este implicit derivată din clasa predefinită *object* (tot una cu *System.Object*). C# nu permite moştenire multiplă, eliminând astfel complicaţiile întâlnite în C++. Ca alternativă, se permite totuşi implementarea de mai multe interfeţe (la fel ca în Java).

5.2.1 Specificarea moştenirii

În C# se pentru o clasă D se defineşte clasa de bază B folosind următoarea formulă:

```
class D: B
{
  //declaratii
}
```

Dacă pentru o anumită clasă nu se specifică două puncte urmate de numele unei clase de bază atunci *object* va deveni bază pentru clasa în cauză.

Exemplu:

```
//clasa de baza in C#
public class Employee
{
  protected string name;
  protected string ssn;
}

//clasa derivata in C#
public class Salaried : Employee
{
  protected double salary;
  public Salaried( string name, string ssn, double salary )
  {
    this.name = name;
    this.ssn = ssn;
    this.salary = salary;
  }
}
```

Se observă că câmpurile *name* şi *ssn* din clasa de bază sunt accesibile în clasa derivată, datorită specificatorului de acces *protected*.

5.2.2 Apelul constructorilor din clasa de bază

În exemplul anterior nu s–a definit nici un constructor în clasa de bază *Employee*; constructorul clasei derivate trebuie să facă iniţializările câmpurilor în conformitate cu parametrii transmişi, chiar dacă o parte din aceste câmpuri provin din clasa de bază. Mai logic ar fi ca în clasa de bază să se găsească un constructor care să iniţializeze câmpurile proprii: *name* şi *ssn*. Întrucât constructorii nu se moştenesc, e nevoie ca în clasa derivată să se facă un apel explicit al constructorului clasei de bază. Acest apel se face prin *iniţializator de constructor* care are forma: două puncte urmate de *base(parametrii-efectivi)*.

```
public class Employee
{
  protected string name;
  protected string ssn;
  public Employee( string name, string ssn)
  {
    this.name = name;
    this.ssn = ssn;
    System.Console.WriteLine(‘‘Employee constructor: {0}, {1}’’,
    name, ssn);
  }
}
public class Salaried : Employee
{
  protected double salary;
  public Salaried(string name, string ssn, double salary):
    base(name, ssn)
  {
    this.salary = salary;
    System.Console.WriteLine(‘‘Salaried constructor: {0}’’,
      salary);
  }
}
class Test
{
  Salaried s = new Salaried(‘‘Jesse’’, ‘‘1234567890’’,
    100000.00);
}
```

La rulare se va obţine:

```
Employee constructor: Jesse, 1234567890
Salaried constructor: 100000.00
```

de unde se deduce că apelul de constructor de clasă de bază se face înaintea executării oricăror alte instrucţiuni conţinute în constructorul clasei derivate.

Dacă o clasă de bază nu are definit nici un constructor, atunci se va crea unul implicit (fără parametri). Dacă după un constructor al unei clase derivate nu se specifică un iniţializator de constructor, atunci va fi apelat constructorul implicit (fie creat automat de compilator, fie scris de către programator); dacă nu există nici un constructor implicit în clasa de bază, atunci programatorul trebuie să specifice un constructor din clasa de bază care va fi apelat, împreună cu parametrii adecvaţi.

5.2.3 Operatorii *is* şi *as*

Operatorul *is*

Operatorl *is* este folosit pentru a verifica dacă un anumit obiect este de un anumit tip. Este folosit de obicei înainte operatţiilor de downcasting (conversie explicită de la un tip de bază la unul derivat). Operatorul se foloseşte astfel:

```
instanta is NumeClasa
```

rezultatul acestei operaţii fiind true sau false.

Exemplu:

```
Employee e = ...;
if (e is Salaried)
{
  Salaried s = (Salaried)e;
}
```

În cazul în care s–ar face conversia explicită iar obiectul nu este de tipul la care se face conversia ar rezulta o excepţie: `System.InvalidCastException`.

Operatorul *as*

Acest operator este folosit pentru conversii explicite, returnând un obiect de tipul la care se face conversia sau null dacă conversia nu se poate face (nu se aruncă excepţii). Determinarea validităţii conversiei se face testând valoarea rezultată faţă de null: dacă rezultatul e null atunci conversia nu s–a putut face. Ca şi precedentul operator se foloseşte în special la downcasting.

Exemplu:

```
Employee e = ...;
Salaried s = e as Salaried;
if (s != null)
{
//se lucreaza cu instanta valida de tip Salaried
}
```

5.3 Clase *sealed*

Specificatorul *sealed* care se poate folosi înaintea cuvântului cheie *class* specifică faptul că clasa curentă nu se poate deriva. Este o eroare de compilare ca o clasă *sealed* să fie declarată drept clasă de bază.

5.4 Polimorfismul

Polimorfismul este capacitatea unei entităţi de a lua mai multe forme. În limbajul C# polimorfismul este de 3 feluri: parametric, ad–hoc şi de moştenire.

5.4.1 Polimorfismul parametric

Este cea mai slabă formă de polimorfism, fiind regăsită în majoritatea altor limbaje, ce nu sunt orientate pe obiecte: Pascal, C. Prin polimorfismul parametric se permite ca o implementare de funcţie să poată prelucra orice număr de parametri. Acest lucru se poate obţine prin folosirea unui parametru de tip *params* (vezi 3.2).

5.4.2 Polimorfismul ad–hoc

Se mai numeşte şi supraîncărcarea metodelor, mecanism prin care în cadrul unei clase se pot scrie mai multe metode, având acelaşi nume, dar tipuri şi numere diferite de parametri de apel. Alegerea funcţiei care va fi apelată se va face la compilare, pe baza corespondenţei între tipurile de apel şi cele formale.

5.4.3 Polimorfismul de moştenire

Este forma cea mai evoluată de polimorfism. Dacă precedentele forme de polimorfism sunt aplicabile fără a se pune problema de moştenire, în acest caz este necesar să existe o ierarhie de clase. Mecanismul se bazează pe faptul că o clasă de bază defineşte un tip care este compatibil din punct de vedere al atribuirii cu orice tip derivat, ca mai jos:

```
class B{...}
class D: B{...}
class Test
{
  static void Main()
  {
    B b = new D();//upcasting=conversie implicita catre baza
  }
}
```

Într–un astfel de caz se pune problema: ce se întâmplă cu metodele având aceeaşi listă de parametri formali şi care se regăsesc în cele două clase?

Să considerăm exemplul următor: avem o clasă *Shape* care conţine o metodă *public void Draw()*; din *Shape* se derivează clasa *Polygon* care implementează aceeaşi metodă în mod specific. Problema care se pune este cum se rezolvă un apel al metodei *Draw* în context de upcasting:

```
class Shape
{
  public void Draw()
  {
    System.Console.WriteLine(‘‘Shape.Draw()’’);
  }
}
class Polygon: Shape
{
  public void Draw()
  {
    System.Console.WriteLine(‘‘Polygon.Draw()’’);
    //desenarea s-ar face prin GDI+
  }
}
class Test
{
  static void Main()
  {
    Polygon p = new Polygon();
    Shape s = p;//upcasting
    s.Draw();
    p.Draw();
  }
}
```

La compilarea acestui cod se va obţine un avertisment:

```
warning CS0108: The keyword new is required on Polygon.Draw()
because it hides inherited member Shape.Draw()
```

dar despre specificatorul *new* vom vorbi mai jos (oricum, adăugarea lui nu va schimba cu nimic comportamentul de mai jos, doar va duce la dispariţia de avertisment). Codul de mai sus va afişa:

```
Shape.Draw()
Polygon.Draw()
```

Dacă cea de–a doua linie afişată este conformă cu intuiţia, primul rezultat este discutabil, dar justificabil: apelul de metodă *Draw()* este rezolvat în fiecare caz la compilare pe baza tipului declarat al obiectelor; ca atare apelul precedent este legat de corpul metodei *Draw* din clasa *Shape*, chiar dacă *s* a fost instanţiat de fapt pe baza unui obiect de tip *Polygon*.

Este posibil ca să se dorească schimbarea acestui comportament: apelul de metodă *Draw* să fie rezolvat în funcţie de tipul efectiv al obiectului care face acest apel, şi nu de tipul formal declarat. În cazul precedent, apelul *s.Draw()* trebuie să se rezolve de fapt ca fiind către metoda *Draw()* din *Polygon*, pentru că acesta este tipul la rulare al obiectului *s*. Cu alte cuvinte, apelul ar trebui să fie rezolvat la rulare şi nu la compilare, în funcţie de natura obiectelor. Acest comportament polimorfic este referit sub denumirea *polimorfism de moştenire*.

5.4.4 *Virtual* şi *override*

Pentru a asigura faptul că legarea apelului de metode se face la rulare şi nu la compilare, e necesar ca în clasa de bază să se specifice că metoda *Draw()* este virtuală, iar în clasa derivată pentru aceeaşi metodă trebuie să se spună că este o suprascriere a celei din bază:

```
class Shape{
  public virtual void Draw(){...}
}
class Polygon{
  public override void Draw(){...}
}
```

În urma executării metodei *Main* din clasa de mai sus, se va afişa:

```
Polygon.Draw()
Polygon.Draw()
```

adică s–a apelat metoda corespunzătoare tipului efectiv de la rulare, în fiecare caz.

În cazul în care clasa *Polygon* este la rândul ei moştenită şi se doreşte ca polimorfismul să funcţioneze în continuare va trebui ca în această a treia clasă să suprascrie (*override*) metoda *Draw()*.

Un astfel de comportament polimorfic este benefic atunci când se foloseşte o colecţie de obiecte de tipul unei clase de bază:

```
Shape[] painting = new Shape[10];
painting[0] = new Shape();
```

```
painting[1] = new Polygon();
...
foreach( Shape s in painting)
  s.Draw();
```

5.4.5 Modificatorul *new* pentru metode

Modificatorul *new* se foloşeste pentru a indica faptul că o metodă dintr-o clasă derivată care are aceeaşi semnătură cu una dintr–o clasă de bază nu este o suprascriere a ei, ci o nouă metodă. Este ca şi cum metoda declarată *new* ar avea o semnătură diferită.

Să presupunem următorul scenariu: compania A crează o clasă *A* care are forma:

```
public class A{
  public void M(){
    Console.WriteLine(‘‘A.M()’’);
  }
}
```

O altă companie B va crea o clasă *B* care moşteneşte clasa A. Compania B nu are nici o influenţă asupra companiei A (sau asupra modului în care aceasta va face modificări asupra clasei *A*). Ea va defini în interiorul clasei B o metodă *M()* şi una *N()*:

```
class B: A{
  public void M(){
    Console.WriteLine(‘‘B.M()’’);
    N();
    base.M();
  }
  protected virtual void N(){
    Console.WriteLine(‘‘B.N()’’);
  }
}
```

Atunci când compania B compilează codul, compilatorul C# va produce următorul avertisment:

```
warning CS0108: The keyword new is required on ‘B.M()’ because
it hides inherited member ‘A.M()’
```

Acest avertisment va notifica programatorul că clasa *B* defineşte o metodă *M()*, care va ascunde metoda *M()* din clasa de bază *A*. Această nouă metodă ar putea schimba înţelesul (semantica) lui *M()*, aşa cum a fost creat iniţial de compania A. Este de dorit în astfel de cazuri compilatorul să avertizeze despre posibile nepotriviri semantice. Programatorii din B vor trebui să pună în orice caz specificatorul *new* înaintea metodei *B.M()*.

Să presupunem că o aplicaţie foloseşte clasa *B()* în felul următor:

```
class App{
  static void Main(){
    B b = new B();
    b.M();
  }
}
```

La rulare se va afişa:

```
B.M()
B.N()
A.M()
```

Să presupunem că A decide adăugarea unei metode virtuale *N()* în clasa sa, metodă ce va fi apelată din *M()*:

```
public class A
{
  public void M()
  {
    Console.WriteLine(‘‘A.M()’’);
    N();
  }
  protected virtual void N()
  {
    Console.WriteLine(‘‘A.N()’’);
  }
}
```

La o recompilare făcută de B, este dat următorul avertisment:

```
warning CS0114: ‘B.N()’ hides inherited member ‘A.N()’. To make
the current member override that implementation, add the
override keyword. Otherwise, add the new keyword.
```

În acest mod compilatorul avertizează că ambele clase oferă o metodă *N()* a căror semantică poate să difere. Dacă B decide că metodele *N()* nu sunt semantic legate în cele două clase, atunci va specifica *new*, informând compilatorul de faptul că versiunea sa este una nouă, care nu suprascrie polimorfic metoda din clasa de bază.

Atunci când codul din clasa *App* este rulat, se va afişa la ieşire:

```
B.M()
B.N()
A.M()
A.N()
```

Ultima linie afişată se explică tocmai prin faptul că metoda *N()* din *B* este declarată *new* şi nu *override* (dacă ar fi fost override, ultima linie ar fi fost *B.N()*, din cauza polimorfismului).

Se poate ca B să decidă că metodele *M()* şi *N()* din cele două clase sunt legate semantic. În acest caz, ea poate şterge definiţia metodei *B.M*, iar pentru a semnala faptul că metoda *B.N()* suprascrie metoda omonimă din clasa părinte, va înlocui cuvântul *new* cu *override*. În acest caz, metoda *App.Main* va produce:

```
A.M()
B.N()
```

ultima linie fiind explicată de faptul că *B.N()* suprascrie o metodă virtuală.

5.4.6 Metode *sealed*

O metodă de tip *override* poate fi declarată ca fiind de tip *sealed*, astfel împiedicându–se suprascrierea ei într–o clasă derivată din cea curentă:

```
using System;
class A
{
  public virtual void F()
  {
    Console.WriteLine(‘‘A.F()’’);
  }
  public virtual void G()
  {
    Console.WriteLine(‘‘A.G()’’);
  }
}
```

```
class B: A
{
  sealed override public void F()
  {
    Console.WriteLine(‘‘B.F()’’);
  }
  override public void G()
  {
    Console.WriteLine(‘‘B.G()’’);
  }
}
class C: B
{
  override public void G()
  {
    Console.WriteLine(‘‘C.G()’’);
  }
}
```

Modificatorul *sealed* pentru $B.F$ va împiedica tipul C să suprascrie metoda F.

5.4.7 Exemplu folosind *virtual, new, override, sealed*

Să presupunem următoare ierarhie de clase, reprezentată în Fig. 5.1; o clasă X moşteneste o clasă Y dacă sensul săgeţii este de la X la Y. Fiecare clasă are o metodă *void f()* care determină afişarea clasei în care este definită şi pentru care se vor specifica *new, virtual, override, sealed*. Să presupunem că clasa de test arată astfel:

```
public class Test
{
  static void Main()
  {
    A[] x = new A[4];
    x[0] = new A();
    x[1] = new B();
    x[2] = new C();
    x[3] = new D();

    A a = new A();
    B b = new B();
```

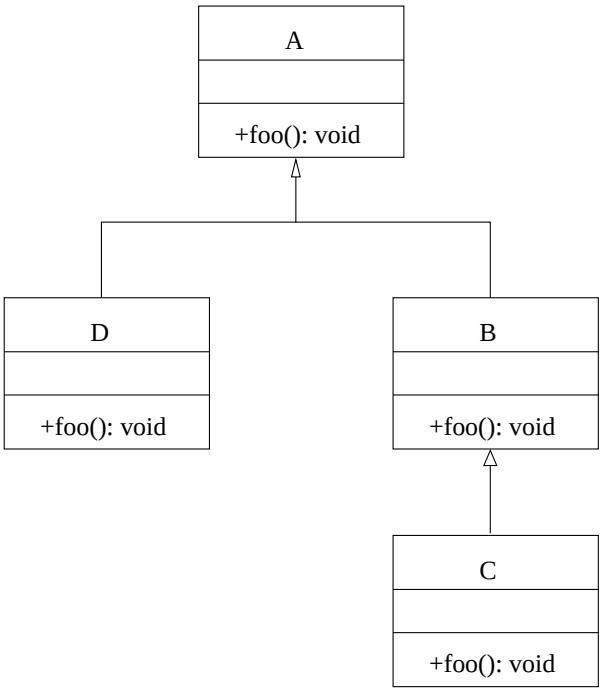

Figura 5.1: Ierarhie de clase

```
    C c = new C();
    D d = new D();

    /* secventa 1 */
    for(int i=0; i<4; i++)
    {
      x[i].f();
    }
    /* secventa 2 */
    a.f();
    b.f();
    c.f();
    d.f();
  }
}
```

În funcţie de specificatorii metodelor *f()* din fiecare clasă, se obţin ieşirile din tabelul 5.1:

Tabelul 5.1: Efecte ale diferiţilor specificatori.

Metoda	A.f()	B.f()	C.f()	D.f()
Specificator	virtual	override	override	override
Ieşire secv. 1 Ieşire secv. 2	A.f A.f	B.f B.f	C.f C.f	D.f D.f
Specificator	virtual	override	new	override
Ieşire secv. 1 Ieşire secv. 2	A.f A.f	B.f B.f	B.f C.f	D.f D.f
Specificator	virtual	new	new	new
Ieşire secv. 1 Ieşire secv. 2	A.f A.f	A.f B.f	A.f C.f	A.f D.f
Specificator	virtual	new	override	override
Eroare de compilare deoarece C.f nu poate suprascrie metoda nevirtuală B.f()				
Specificator	virtual	virtual	override	override
Ieşire secv. 1 Ieşire secv. 2 Avertisment la compilare deoarece B.f înlocuieşte A.f	A.f A.f	A.f B.f	A.f C.f	D.f D.f
Specificator	virtual	sealed override	override	override
Eroare de compilare deoarece deoarece B.f nu poate fi suprascrisă de C.f				

5.5 Clase şi metode abstracte

Deseori pentru o anumită clasă nu are sens crearea de instanţe, din cauza unei generalităţi prea mari a tipului respectiv. Spunem că această clasă este *abstractă*, iar pentru a împiedica efectiv crearea de instanţe de acest tip, se va specifica cuvântul *abstract* înaintea metodei. În exemplele de mai sus, clasele *Employee* şi *Shape* ar putea fi gândite ca fiind abstracte: ele conţin prea puţină informaţie pentru a putea crea instanţe utile.

Analog, pentru o anumită metodă din interiorul unei clase uneori nu se poate specifica o implementare. De exemplu, pentru clasa *Shape* de mai sus,

este imposibil să se dea o implementare la metoda *Draw()*, tocmai din cauza generalităţii acestei clase. Ar fi util dacă pentru această metodă programatorul ar fi obligat să dea implementări specifice ale acestei metode pentru diversele clase derivate. Pentru a se asigura tratarea polimorfică a acestui tip abstract, orice metodă abstractă este automat şi virtuală. Orice metodă care este declarată abstractă implică declararea clasei ca fiind abstractă.

Exemplu:

```
abstract class Shape
{
  public abstract void Draw();
  //de remarcat lipsa implementarii si semnul punct si virgula
}
```

Orice clasă care este derivată dintr–o clasă abstractă va trebui fie să nu aibă nici o metodă abstractă moştenită fără implementare, fie să se declare ca fiind abstractă. Existenţa de metode neimplementate nu va permite instanţierea clasei respective.

5.6 Interfeţe

O interfaţă defineşte un contract. O clasă sau o structură care implementează o interfaţă aderă la acest contract. Relaţia dintre o interfaţă şi un tip care o implementează este deosebită de cea existentă între clase (este un/o): este o relaţie de implementare.

O interfaţă poate conţine metode, proprietăţi, evenimente, indexatori. Ea însă nu va conţine implementări pentru acestea, doar declaraţii. Declararea unei interfeţe se face astfel:
atribute$_{opt}$ modificatori-de-interfaţă$_{opt}$ `interface` identificator baza-interfeţei$_{opt}$ corp-interfaţă ;$_{opt}$
Modificatorii de interfaţă sunt: *new, public, protected, internal, private.* O interfaţă poate să moştenească de la zero sau mai multe interfeţe. Corpul interfeţei conţine declaraţii de metode, fără implementări. Orice metodă are gradul de acces public. Nu se poate specifica pentru o metodă din interiorul unei interfeţe: *abstract, public, protected, internal, private, virtual, override*, ori *static*.

Exemplu:

```
interface IStorable
{
  void Read( );
```

```
  void Write();
}
```

O clasă care implementează o astfel de interfaţă se declară ca mai jos:

```
class Document: IStorable
{
  public void Read(){/*cod*/}
  public void Write(){/*cod*/}
  //alte declaratii
}
```

O clasă care implementează o interfaţă trebuie să definească toate metodele care se regăsesc în interfaţa respectivă. Următoarele reguli trebuie respectate la implementarea de interfeţe:

1. Tipul de retur al metodei din clasă trebuie să coincidă cu tipul de retur al metodei din interfaţă

2. Tipul parametrilor formali din metodă trebuie să fie acelaşi cu tipul parametrilor formali din interfaţă

3. Metoda din clasă trebuie să fie declarată publică şi nestatică.

Aceste implementări pot fi declarate folosind specificatorul *virtual* (deci subclasele clasei curente pot folosi *new* şi *override*).

Exemplu:

```
using System;
interface ISavable
{
  void Read();
  void Write();
}
public class TextFile : ISavable
{
  public virtual void Read()
  {
    Console.WriteLine("TextFile.Read()");
  }
  public void Write()
  {
    Console.WriteLine("TextFile.Write()");
```

```
  }
}
public class ZipFile : TextFile
{
  public override void Read()
  {
    Console.WriteLine("ZipFile.Read()");
  }
  public new void Write()
  {
    Console.WriteLine("ZipFile.Write()");
  }
}
public class Test
{
  static void Main()
  {
    Console.WriteLine("\nTextFile reference to ZipFile");
    TextFile textRef = new ZipFile();
    textRef.Read();
    textRef.Write();
    Console.WriteLine("\nISavable reference to ZipFile");
    ISavable savableRef = textRef as ISavable;
    if(savableRef != null)
    {
      savableRef.Read();
      savableRef.Write();
    }
    Console.WriteLine("\nZipFile reference to ZipFile");
    ZipFile zipRef = textRef as ZipFile;
    if(zipRef!= null)
    {
      zipRef.Read();
      zipRef.Write();
    }
  }
}
```

La ieşire se va afişa:

```
TextFile reference to ZipFile
ZipFile.Read()
```

```
TextFile.Write()

ISavable reference to ZipFile
ZipFile.Read()
TextFile.Write()

ZipFile reference to ZipFile
ZipFile.Read()
ZipFile.Write()
```

În exemplul de mai sus se foloşeste operatorul *as* pentru a obţine o referinţă la interfeţe, pe baza obiectelor create. În general, se preferă ca apelul metodelor care sunt implementate din interfaţă să se facă via o referinţă la interfaţa respectivă, obţinută prin intermediul operatorului *as* (ca mai sus) sau după o testare prealabilă prin *is* urmată de conversie explicită, ca mai jos:

```
if (textRef is ISavable)
{
  ISavable is = (ISavable)textRef;
  is.Read();//etc
}
```

În general, dacă se doreşte doar răspunsul la întrebarea "este obiectul curent un implementator al interfeţei I?", atunci se recomandă folosirea operatorului *is*. Dacă se ştie că va trebui făcută şi o conversie la tipul interfaţă, atunci este mai eficientă folosirea lui *as*. Afirmaţia se bazează pe studiul codului IL rezultat în fiecare caz.

Să presupunem că exista o interfaţă `I` având metoda `M()` care este implementată de o clasă `C`, care defineşte metoda `M()`. Este posibil ca această metodă să nui aibă o semnificaţie în afara clasei `C`, ca atare a nu e util ca metoda `M()` să fie declarată publică. Mecanismul care permite acest lucru se numeşte *ascunderea numelor*. Această tehnică permite ascunderea metodelor moştenite dintr-o interfaţă, acestea devenind private (calificarea lor ca fiind publice este semnalată ca o eroare).

Metodele din interfeţe care s–au implementat explicit (prin calificarea lor cu numele interfeţei de origine) nu pot fi declarate *abstract, virtual, override, new*. Mai mult, asemenea metode nu pot fi accesate direct prin intermediul unui obiect (*obiect.NumeMetoda*), ci doar prin intermediul unei conversii către interfaţă respectivă, deoarece prin implementare explicită a metodelor aceste devin private şi singura modalitate de acces a lor este upcasting-ul către interfaţă.

Exemplu:

```
using System;
public interface IDataBound
{
  void Bind();
}

public class EditBox : IDataBound
{
  // implementare de IDataBound
  void IDataBound.Bind()
  {
    Console.WriteLine("Binding to data store...");
  }
}

class NameHidingApp
{
  public static void Main()
  {
    Console.WriteLine();
    EditBox edit = new EditBox();
    Console.WriteLine("Apel EditBox.Bind()...");
    //EROARE: Aceasta linie nu se compileaza deoarece metoda
    //Bind nu mai exista ca metoda publica in clasa EditBox
    edit.Bind();
    Console.WriteLine();

    IDataBound bound = (IDataBound)edit;
    Console.WriteLine("Apel (IDataBound) EditBox.Bind()...");
    // Functioneaza deoarece s-a facut conversie la IDataBound
    bound.Bind();
  }
}
```

Este posibil ca un tip să implementeze mai multe interfeţe. Atunci când două interfeţe au o metodă cu aceeaşi semnătură, programatorul are mai multe variante de lucru. Cel mai simplu, el poate să furnizeze o singură implementare pentru ambele metode, ca mai jos:

```
interface IFriendly
{
  void GreetOwner() ;
```

```
}
interface IAffectionate
{
  void GreetOwner() ;
}
abstract class Pet
{
  public virtual void Eat()
  {
    Console.WriteLine( "Pet.Eat" ) ;
  }
}
class Dog : Pet, IAffectionate, IFriendly
{
  public override void Eat()
  {
    Console.WriteLine( "Dog.Eat" ) ;
  }
  public void GreetOwner()
  {
    Console.WriteLine( "Woof!" ) ;
  }
}
```

O altă modalitate este să se specifice implicit care metodă este implementată.

```
class Dog : Pet, IAffectionate, IFriendly
{
  public override void Eat()
  {
    Console.WriteLine( "Dog.Eat" ) ;
  }
  void IAffectionate.GreetOwner()
  {
    Console.WriteLine( "Woof!" ) ;
  }
  void IFriendly.GreetOwner()
  {
    Console.WriteLine( "Jump up!" ) ;
  }
}
public class Pets
```

```
{
  static void Main()
  {
    IFriendly mansBestFriend = new Dog() ;
    mansBestFriend.GreetOwner() ;
    (mansBestFriend as IAffectionate).GreetOwner() ;
  }
}
```

La ieşire se va afişa:

```
Jump up!
Woof!
```

Dacă însă în clasa Dog se adaugă metoda

```
public void GreetOwner()
{
Console.WriteLine( "Woof!" ) ;
}
```

(care a fost iniţial definită), atunci se poate face apel de tipul `dog.GreetOwner()` (dog este instanţă de Dog); apelurile de metode din interfaţă rămân de asemenea valide. Rezultatul este ca şi la exemplul precedent: se afişează mesajul `Woof`.

5.6.1 Clase abstracte sau interfeţe?

Atât interfeţele cât şi clasele abstracte au comportamente similare şi pot fi folosite în situaţii similare. Dar totuşi ele nu se pot substitui reciproc. Câteva principii generale de utilizare a lor sunt date mai jos.

Dacă o relaţie se poate exprima mai degrabă ca "este un/o" decât altfel, atunci entitatea de bază ar trebui gândită ca o clasă abstractă.

Un alt aspect este bazat pe obiectele care ar folosi capabilităţile din tipul de bază. Dacă aceste capabilităţi ar fi folosite de către obiecte care nu sunt legate între ele, atunci ar fi indicată o interfaţă.

Dezavantajul claselor abstracte este că nu poate fi decât bază unică pentru orice altă clasă.

Curs 6

Delegaţi. Evenimente. Structuri

6.1 Tipul delegat

În programare deseori apare următoarea situaţie: trebuie să se execute o anumită acţiune, dar nu se ştie de dinainte care anume, sau chiar ce obiect va trebui efectiv utilizat. De exemplu, un buton poate şti că trebuie să anunţe un obiect despre faptul că fost apăsat, dar nu va şti aprioric pe cine. Mai degrabă decât să se lege butonul de un obiect particular, butonul va declara un delegat, pentru care clasa interesată de evenimentul de apăsare va da o implementare.

Fiecare acţiune pe care utilizatorul o execută pe o interfaţă grafică declanşează un eveniment. Alte evenimente se pot declanşa independent de acţiunile utilizatorului: sosirea unui email, terminarea copierii unor fişiere, sfârşitul unei interogări pe o bază de date, etc. Un eveniment este o încapsulare a ideii că "se întâmplă ceva" la care programul trebuie să răspundă. Evenimentele şi delegaţii sunt strâns legate deoarece răspunsul la acest eveniment se va face de către un event handler, care în C# se implementează ca un delegat.

Un delegat este un tip referinţă folosit pentru a încapsula o metodă cu un anumit antet (tipul parametrilor formali şi tipul de retur). Orice metodă care are acelaşi antet poate fi legată la un anumit delegat. Într–un limbaj precum C++, acest lucru se rezolvă prin intermediul pointerilor la funţii. Delegaţii rezolvă aceeaşi problemă, dar într–o manieră orientată obiect şi cu garanţii asupra siguranţei codului rezultat, precum şi cu o uşoară generalizare (vezi delegaţii multicast).

Un delegat este creat după următoarea sintaxă:

atribute$_{opt}$ modificatori-de-delegat$_{opt}$ `delegate` tip-retur identificator(lista-param-formali$_{opt}$);
Modificatorul de delegat poate fi: *new, public, protected, internal, private.* Un delegat se poate specifica atât în interiorul unei clase, cât şi în exteriorul ei, fiind de fapt o declaraţie de clasă derivată din *System.Delegate*. Dacă este declarat în interiorul unei clase, atunci este şi static (a se vedea statutul claselor interioare).

Exemplu:

```
public delegate int WhichIsFirst(object obj1, object obj2);
```

6.1.1 Utilizarea delegaţilor pentru a specifica metode la runtime

Să presupunem că se doreşte crearea unei clase container simplu numit *Pair* care va conţine două obiecte pentru care va putea face şi sortare. Nu se va şti aprioric care va fi tipul obiectelor conţinute, deci se va folosi pentru ele tipul *object*. Dar sortarea celor două obiecte se va face diferit, în funcţie de tipul lor efectiv: de exemplu pentru nişte persoane (clasa *Student* în cele ce urmează) se va face după nume, pe când pentru animale (clasa *Dog*) se va face după alt criteriu: greutatea. Containerul *Pair* va trebui să facă faţă acestor clase diferite. Rezolvarea se va da prin delegaţi.

Clasa *Pair* va defini un delegat, *WhichIsFirst*. Metoda *Sort* de ordonare va prelua ca (unic) parametru o instanţă a metodei *WhichIsFirst*, care va implementa relaţia de ordine, în funcţie de tipul obiectelor conţinute. Rezultatul unei comparaţii între două obiecte va fi de tipul enumerare *Comparison*, definit de utilizator:

```
public enum Comparison
{
  theFirstComesFirst = 0,
  //primul obiect din colectie este primul in ordinea sortarii
  theSecondComesFirst = 1
  //al doilea obiect din colectie este primul in ordinea sortarii
}
```

Delegatul (tipul de metodă care realizează compararea) se declară astfel:

```
//declarare de delegat
public delegate Comparison WhichIsFirst(
                          object obj1, object obj2);
```

Clasa Pair se declară după cum urmează:

```
public class Pair
{
  //tabloul care contine cele doua obiecte
  private object[] thePair = new object[2];

  //constructorul primeste cele doua obiecte continute
  public Pair( object firstObject, object secondObject)
  {
    thePair[0] = firstObject;
    thePair[1] = secondObject;
  }

  //metoda publica pentru ordonarea celor doua obiecte
  //dupa orice criteriu
  public void Sort( WhichIsFirst theDelegatedFunc )
  {
    if (theDelegatedFunc(thePair[0],thePair[1]) ==
        Comparison.theSecondComesFirst)
    {
      object temp = thePair[0];
      thePair[0] = thePair[1];
      thePair[1] = temp;
    }
  }
  //metoda ce permite tiparirea perechii curente
  //se foloseste de polimorfism - vezi mai jos
  public override string ToString( )
  {
    return thePair[0].ToString()+", "+thePair[1].ToString();
  }
}
```

Clasele Student şi Dog sunt:

```
public class Dog
{
  public Dog(int weight)
  {
    this.weight=weight;
  }
  //Ordinea este data de greutate
  public static Comparison WhichDogComesFirst(
```

```
    Object o1, Object o2)
  {
    Dog d1 = o1 as Dog;
    Dog d2 = o2 as Dog;
    return d1.weight > d2.weight ?
     Comparison.theSecondComesFirst :
     Comparison.theFirstComesFirst;
  }
  //pentru afisarea greutatii unui caine
  public override string ToString( )
  {
    return weight.ToString( );
  }
  private int weight;
}

public class Student
{
  public Student(string name)
  {
    this.name = name;
  }
  //studentii sunt ordonati alfabetic
  public static Comparison WhichStudentComesFirst(
    Object o1, Object o2)
  {
    Student s1 = o1 as Student;
    Student s2 = o2 as Student;
    return (String.Compare(s1.name, s2.name) < 0 ?
      Comparison.theFirstComesFirst :
      Comparison.theSecondComesFirst);
  }
  //pentru afisarea numelui unui student
  public override string ToString( )
  {
    return name;
  }
  private string name;
}
```

Clasa de test este:

```
public class Test
{
  public static void Main( )
  {
    //creaza cate doua obiecte
    //de tip Student si Dog
    //si containerii  corespunzatori
    Student Stacey = new Student(‘‘Stacey’’);
    Student Jesse = new Student (‘‘Jess’’);
    Dog Milo = new Dog(10);
    Dog Fred = new Dog(5);
    Pair studentPair = new Pair(Stacey, Jesse);
    Pair dogPair = new Pair(Milo, Fred);
    Console.WriteLine(‘‘studentPair\t: {0}’’, studentPair);
    Console.WriteLine(‘‘dogPair\t: {0}’’, dogPair);
    //Instantiaza delegatii
    WhichIsFirst theStudentDelegate =
     new WhichIsFirst(Student.WhichStudentComesFirst);
    WhichIsFirst theDogDelegate =
     new WhichIsFirst(Dog.WhichDogComesFirst);
    //sortare folosind delegatii
    studentPair.Sort(theStudentDelegate);
    Console.WriteLine(‘‘Dupa sortarea pe studentPair\t: {0}’’,
     studentPair.ToString( ));
    dogPair.Sort(theDogDelegate);
    Console.WriteLine(‘‘Dupa sortarea pe dogPair\t\t: {0}’’,
     dogPair.ToString( ));
  }
}
```

6.1.2 Delegaţi statici

Unul din aspectele neelegante ale exemplului anterior este că e necesar ca în clasa *Test* să se instanţieze delegaţii care sunt necesari pentru a ordona obiectele din *Pair*. O modalitate mai bună este să se obţină delegaţii direct din clasele *Student* şi *Dog*. Acest lucru se obţine prin crearea unui delegat static în interiorul fiecărei clase:

```
public static readonly WhichIsFirst OrderStudents =
new WhichIsFirst(Student.WhichStudentComesFirst);
```

(analog pentru clasa *Dog*; de remarcat că "static readonly" nu se poate înlocui cu "const", deoarece iniţilizatosul nu este considerat expresie constantă). Declaraţia de mai sus se foloseşte astfel:

```
...
studentpair.Sort(Student.OrderStudent);
...
```

rezultatul fiind identic.

În [2] este dată şi o implementare de delegat ca proprietate statică, aceasta ducând la crearea unui delegat doar în cazul în care este nevoie de el[1].

6.1.3 Multicasting

Uneori este nevoie ca un delegat să poată apela mai mult de o singură metodă. De exemplu, atunci când un buton este apăsat, se poate să vrei să efectuezi mai mult de o sigură acţiune: să scrii într–un textbox un şir de caractere şi să înregistrezi într–un fişier faptul că s–a apăsat acel buton (logging). Acest lucru s–ar putea rezolva prin construirea unui vector de delegaţi care să conţină toate metodele dorite, însă acest lucru ar duce la un cod greu de urmărit şi inflexibil; pentru un astfel de exemplu, a se vedea [2], pag. 261–265. Mult mai simplu ar fi dacă un delegat ar putea să conţină mai mult de o singură metodă. Acest lucru se numeşte *multicasting* şi este folosit intens la tratarea evenimentelor (vezi secţiunea următoare).

Orice delegat care returnează *void* este un delegat multicast, care poate fi tratat şi ca un delegat single-cast. Doi delegaţi multicast pot fi combinaţi folosind semnul +. Rezultatul unei astfel de "adunări" este un nou delegat multicast care la apelare va invoca metodele conţinute, în ordinea în care s–a făcut adunarea. De exemplu, dacă *Writer* şi *Logger* sunt delegaţi care returnează void, atunci următoarea linie va produce combinarea lor într–un singur delegat:

```
myMulticastDelegate = Writer + Logger;
```

Se pot adăuga delegaţi multicast folosind operatorul +=, care va adăuga delegatul de la dreapta operatorului la delegatul multicast aflat în stânga sa:

```
myMulticastDelegate += Transmitter;
```

presupunând că Transmitter este compatibil cu *myMulticastDelegate* (are aceeaşi semnătură). Operatorul $-=$ funcţionează invers faţă de $+=$.

Exemplu:

[1] Tehnică numită iniţializaer târzie (lazy initialization)

```
using System;
//declaratia de delegat multicast
public delegate void StringDelegate(string s);

public class MyImplementingClass
{
  public static void WriteString(string s)
  {
    Console.WriteLine("Writing string {0}", s);
  }
  public static void LogString(string s)
  {
    Console.WriteLine("Logging string {0}", s);
  }
  public static void TransmitString(string s)
  {
    Console.WriteLine("Transmitting string {0}", s);
  }
}

public class Test
{
  public static void Main( )
  {
    //defineste trei obiecte delegat
    StringDelegate Writer,
      Logger, Transmitter;
    //defineste alt delegat
    //care va actiona ca un delegat multicast
    StringDelegate myMulticastDelegate;
    //Instantiaza primii trei delegati
    //dand metodele ce se vor incapsula
    Writer = new StringDelegate(
      MyImplementingClass.WriteString);
    Logger = new StringDelegate(
      MyImplementingClass.LogString);
    Transmitter = new StringDelegate(
      MyImplementingClass.TransmitString);
    //Invoca metoda delegat Writer
    Writer("String passed to Writer\n");
    //Invoca metoda delegat Logger
```

```
    Logger("String passed to Logger\n");
    //Invoca metoda delegat Transmitter
    Transmitter("String passed to Transmitter\n");
    //anunta utilizatorul ca va combina doi delegati
    Console.WriteLine(
      "myMulticastDelegate = Writer + Logger");
    //combina doi delegati, rezultatul este
    //asignat lui myMulticastDelagate
    myMulticastDelegate = Writer + Logger;
    //apelaeaza myMulticastDelegate
    //de fapt vor fi chemate cele doua metode
    myMulticastDelegate(
      "First string passed to Collector");
    //Anunta utilizatorul ca se va adauga al treilea delegat
    Console.WriteLine(
      "\nmyMulticastDelegate += Transmitter");
    //adauga al treilea delegat
    myMulticastDelegate += Transmitter;
    //invoca cele trei metode delagate
    myMulticastDelegate(
      "Second string passed to Collector");
    //anunta utilizatorul ca se va scoate delegatul Logger
    Console.WriteLine(
      "\nmyMulticastDelegate -= Logger");
    //scoate delegatul Logger
    myMulticastDelegate -= Logger;
    //invoca cele doua metode delegat ramase
    myMulticastDelegate(
      "Third string passed to Collector");
  }
}
```

La ieşire vom avea:

```
Writing string String passed to Writer
Logging string String passed to Logger
Transmitting string String passed to Transmitter
myMulticastDelegate = Writer + Logger
Writing string First string passed to Collector
Logging string First string passed to Collector
myMulticastDelegate += Transmitter
Writing string Second string passed to Collector
```

```
Logging string Second string passed to Collector
Transmitting string Second string passed to Collector
myMulticastDelegate -= Logger
Writing string Third string passed to Collector
Transmitting string Third string passed to Collector
```

6.2 Evenimente

Interfeţele grafice actuale cer ca un anumit program să răspundă la *evenimente*. Un eveniment poate fi de exemplu apăsarea unui buton, terminarea transferului unui fişier, selectarea unui meniu, etc; pe scurt, se întâmplă ceva la care trebuie să se dea un răspuns. Nu se poate prezice ordinea în care se petrec evenimentele, iar la apariţia unuia se va cere tratarea sa.

Alte clase pot fi interesate în a răspunde la aceste evenimente. Modul în care vor răspunde va fi extrem de particular, iar clasa care generează evenimentul (ex: clasa Button, la apăsarea unui buton) nu trebuie să ştie modul în care se va răspunde. Butonul va comunica faptul că a fost apăsat, iar clasele interesate în acest eveniment vor reacţiona în consecinţă.

6.2.1 Publicarea şi subscrierea

În C#, orice obiect poate să *publice* un set de evenimente la care alte clase pot să *subscrie*. Când obiectul care a publicat evenimentul îl şi semnaleazaă, toate obiectele care au subscris la acest eveniment sunt notificate. În acest mod se defineşte o dependenţă de tip *one–to–many* între obiecte astfel încât un obiect îşi schimbă starea, toate celelate obiecte dependente sunt notificate şi modificate automat.

De exemplu, un buton poate să notifice un număr oarecare de observatori atunci când a fost apăsat. Butonul va fi numit *publicator*[2] deoarece publică evenimentul *Click* iar celelalte clase sunt numite *abonaţi*[3] deoarece ele subscriu la evenimentul Click.

6.2.2 Evenimente şi delegaţi

Tratarea evenimentelor în C# se face folosind delegaţi. Clasa ce publică defineşte un delegat pe care clasele abonate trebuie să îl implementeze. Când evenimentul este declanşat, metodele claselor abonate vor fi apelate prin

[2]Engl: publisher
[3]Engl: subscribers

intermediul delegatului (pentru care se prevede posibilitatea de a fi multicast, astfel încât să se permită mai mulţi abonaţi).

Metodele care răspund la un eveniment se numesc *event handlers*. Prin convenţie, un event handler în .NET Framework returnează *void* şi preia doi parametri: primul parametru este sursa evenimentului (obiectul publicator); al doilea parametru are tip *EventArgs* sau derivat din acesta.

Declararea unui eveniment se face astfel:
atribute$_{opt}$ modificatori-de-eveniment$_{opt}$ `event` tip nume–eveniment
Modificator-de-eveniment poate fi *abstract, new, public, protected, internal, private, static, virtual, sealed, override, extern*. *Tip* este un handler de eveniment (delegat multicast).

Exemplu:

```
public event SecondChangeHandler OnSecondChange;
```

Vom da mai jos un exemplu care va construi următoarele: o clasă *Clock* care foloseşte un eveniment (*OnSecondChange*) pentru a notifica potenţialii abonaţi atunci când timpul local se schimbă cu o secundă. Tipul acestui eveniment este un delegat *SecondChangeHandler* care se declară astfel:

```
public delegate void SecondChangeHandler(
  object clock, TimeInfoEventArgs timeInformation );
```

în conformitate cu metodologia de declarare a unui event handler, pomenită mai sus. Tipul *TimeInfoEventArgs* este definit de noi ca o clasă derivată din *EventArgs*:

```
public class TimeInfoEventArgs : EventArgs
{
  public TimeInfoEventArgs( int hour, int minute, int second )
  {
    this.hour = hour;
    this.minute = minute;
    this.second = second;
  }
  public readonly int hour;
  public readonly int minute;
  public readonly int second;
}
```

Această clasă va conţine informaţie despre timpul curent. Informaţia este accesibilă *readonly*.

Clasa *Clock* va conţine o metodă *Run()*:

```
public void Run()
{
  for(;;)
  {
    //dormi 10 milisecunde
    Thread.Sleep(10);
    //obtine timpul curent
    System.DateTime dt = System.DateTime.Now();
    //daca timpul s-a schimbat cu o secunda
    //atunci notifica abonatii
    if( dt.Second != second)
    //second este camp al clasei Clock
    {
      //creeaza obiect TimeInfoEventArgs
      //ce va fi transmis abonatilor
      TimeInfoEventArgs timeInformation =
        new TimeInfoEventArgs(dt.Hour, dt.Minute, dt.Second);
      //daca cineva este abonat, atunci anunta-l
      if (OnSecondChange != null)
      {
        OnSeconChange(this, timeInformation);
      }
    }
    //modifica timpul curent in obiectul Clock
    this.second = dt.Second;
    this.minute = dt.Minute;
    this.hour = dt.Hour;
  }
}
```

Metoda *Run* creează un ciclu infinit care interoghează periodic ceasul sistem. Dacă timpul s–a schimbat cu o secundă faţă de timpul precedent, se vor notifica toate obiectele abonate după care îşi va modifica starea, prin cele trei atribuiri finale.

Tot ce rămâne de făcut este să se scrie nişte clase care să subscrie la evenimentul publicat de clasa *Clock*. Vor fi două clase: una numită *DisplayClock* care va afişa pe ecran timpul curent şi o alta numită *LogCurrentTime* care ar trebui să înregistreze evenimentul într–un fişier, dar pentru simplitate va afişa doar la dispozitivul curent de ieşire informaţia tranmsisă:

```
public class DisplayClock
{
```

```
  public void Subscribe(Clock theClock)
  {
    theClock.OnSecondChange +=
      new Clock.SecondChangeHandler(TimeHasChanged);
  }
  void TimeHasChanged(
    object theClock, TimeInfoEventArgs ti)
  {
    Console.WriteLine("Current Time: {0}:{1}:{2}",
      ti.hour.ToString( ),
      ti.minute.ToString( ),
      ti.second.ToString( ));
  }
}
public class LogCurrentTime
{
  public void Subscribe(Clock theClock)
  {
    theClock.OnSecondChange +=
      new Clock.SecondChangeHandler(WriteLogEntry);
  }
  //Aceasta metoda ar trebui sa scrie intr-un fisier
  //dar noi vom scrie la consola
  void WriteLogEntry(
    object theClock, TimeInfoEventArgs ti)
  {
    Console.WriteLine("Logging to file: {0}:{1}:{2}",
    ti.hour.ToString( ),
    ti.minute.ToString( ),
    ti.second.ToString( ));
  }
}
```

De remarcat faptul că evenimentele sunt adăugate folosind operatorul +=.

Exemplul în întregime este dat mai jos:

```
using System;
using System.Threading;
//o clasa care va contine informatie despre eveniment
//in acest caz va contine informatie disponibila in clasa Clock
public class TimeInfoEventArgs : EventArgs
{
```

```
  public TimeInfoEventArgs(int hour, int minute, int second)
  {
    this.hour = hour;
    this.minute = minute;
    this.second = second;
  }
  public readonly int hour;
  public readonly int minute;
  public readonly int second;
}
//clasa care publica un eveniment: OnSecondChange
//clasele care se aboneaza vor subscrie la acest eveniment
public class Clock
{
  //delegatul pe care abonatii trebuie sa il implementeze
  public delegate void SecondChangeHandler(
    object clock, TimeInfoEventArgs timeInformation );
  //evenimentul ce se publica
  public event SecondChangeHandler OnSecondChange;
  //ceasul este pornit si merge la infinit
  //va declansa un eveniment pentru fiecare secunda trecuta
  public void Run( )
  {
    for(;;)
    {
      //inactiv 10 ms
      Thread.Sleep(10);
      //citeste timpul curent al sistemului
      System.DateTime dt = System.DateTime.Now;
      //daca s-a schimbat fata de secunda anterior inregistrata
      //atunci notifica pe abonati
      if (dt.Second != second)
      {
        //creaza obiectul TimeInfoEventArgs
        //care va fi transmis fiecarui abonat
        TimeInfoEventArgs timeInformation =
          new TimeInfoEventArgs(
            dt.Hour,dt.Minute,dt.Second);
        //daca cineva a subscris la acest eveniment
        //atunci anunta-l
        if (OnSecondChange != null)
```

```
        {
          OnSecondChange(this,timeInformation);
        }
      }
      //modifica starea curenta
      this.second = dt.Second;
      this.minute = dt.Minute;
      this.hour = dt.Hour;
    }
  }
  private int hour;
  private int minute;
  private int second;
}
//un observator (abonat)
//DisplayClock va subscrie la evenimentul lui Clock
//DisplayClock va afisa timpul curent
public class DisplayClock
{
  //dandu-se un obiect clock, va subscrie
  //la evenimentul acestuia
  public void Subscribe(Clock theClock)
  {
    theClock.OnSecondChange +=
      new Clock.SecondChangeHandler(TimeHasChanged);
  }
  //handlerul de eveniment de pe partea
  //clasei DisplayClock
  void TimeHasChanged(
    object theClock, TimeInfoEventArgs ti)
  {
    Console.WriteLine("Current Time: {0}:{1}:{2}",
    ti.hour.ToString( ),
    ti.minute.ToString( ),
    ti.second.ToString( ));
  }
}
//un al doilea abonat care ar trebui sa scrie intr-un fisier
public class LogCurrentTime
{
  public void Subscribe(Clock theClock)
```

```
  {
    theClock.OnSecondChange +=
    new Clock.SecondChangeHandler(WriteLogEntry);
  }
  //acest handler ar trebui sa scrie intr-un fisier
  //dar va scrie la standard output
  void WriteLogEntry(
    object theClock, TimeInfoEventArgs ti)
  {
    Console.WriteLine("Logging to file: {0}:{1}:{2}",
    ti.hour.ToString( ),
    ti.minute.ToString( ),
    ti.second.ToString( ));
  }
}
public class Test
{
  static void Main( )
  {
    //creaza un obiect de tip Clock
    Clock theClock = new Clock( );
    //creaza un obiect DisplayClock care
    //va subscrie la evenimentul obiectului
    //Clock anterior creat
    DisplayClock dc = new DisplayClock( );
    dc.Subscribe(theClock);
    //analog se creeaza un obiect de tip LogCurrentTime
    //care va subscrie la acelasi eveniment
    //ca si obiectul DisplayClock
    LogCurrentTime lct = new LogCurrentTime( );
    lct.Subscribe(theClock);
    //porneste ceasul
    theClock.Run( );
  }
}
```

La ieşire se va afişa:

```
Current Time: 14:53:56
Logging to file: 14:53:56
Current Time: 14:53:57
Logging to file: 14:53:57
```

```
Current Time: 14:53:58
Logging to file: 14:53:58
Current Time: 14:53:59
Logging to file: 14:53:59
```

6.2.3 Comentarii

S–ar putea pune următoarea întrebare: de ce este nevoie de o astfel de redirectare de eveniment, când în metoda *Run()* se poate afişa direct pe ecran sau într–un fişier informaţia cerută? Avantajul abordării anterioare este că se pot crea oricâte clase care să fie notificate atunci când acest eveniment se declanşează. Clasele abonate nu trebuie să ştie despre modul în care lucrează clasa *Clock*, iar clasa *Clock* nu trebuie să ştie despre clasele care vor subscrie la evenimentul său. Similar, un buton poate să publice un eveniment *OnClick* şi orice număr de obiecte pot subscrie la acest eveniment, primind o notificare atunci când butonul este apăsat.

Publicatorul şi abonaţii sunt decuplaţi. Clasa *Clock* poate să modifice modalitatea de detectare a schimbării de timp fără ca acest lucru să impună o schimbare în clasele abonate. De asemenea, clasele abonate pot să îşi modifice modul de tratare a evenimentului, în mod transparent faţă de clasa *Clock*. Toate aceste caracteristici fac întreţinerea codului extrem de facilă.

6.3 Structuri

Structurile reprezintă tipuri de date asemănătoare claselor, cu principala diferenţă că sunt tipuri valoare (o astfel de variabilă va conţine direct valoarea, şi nu o adresă de memorie). Sunt considerate versiuni "uşoare" ale claselor, sunt folosite predilect pentru tipuri pentru care aspectul comportamental este mai puţin pronunţat.

Declaraţia unei structuri se face astfel:
atribute$_{opt}$ modificatori-de-struct$_{opt}$ `struct` identificator :interfeţe$_{opt}$ corp ;$_{opt}$
Modificatorii de structură sunt: *new, public, protected, internal, private*. O structură este automat derivată din *System.ValueType*, care la rândul ei este derivată din *System.Object*; de asemenea, este automat considerată *sealed* (sect. 5.3). Poate însă să implementeze una sau mai multe interfeţe.

O structură poate să conţină declaraţii de constante, câmpuri, metode, proprietăţi, evenimente, indexatori, operatori, constructori, constructori statici, tipuri interioare. Nu poate conţine destructor.

La atribuire, se face o copiere a valorilor conţinute de către sursă în destinaţie (indiferent de tipul câmpurilor: valoare sau referinţă).

Exemplu:

```
using System;
public struct Point
{
  public Point(int xCoordinate, int yCoordinate)
  {
    xVal = xCoordinate;
    yVal = yCoordinate;
  }
  public int X
  {
    get
    {
      return xVal;
    }
    set
    {
      xVal = value;
    }
  }
  public int Y
  {
    get
    {
      return yVal;
    }
    set
    {
      yVal = value;
    }
}

  public override string ToString( )
  {
    return (String.Format(``{0}, {1}'', xVal,yVal));
  }
  public int xVal;
  public int yVal;
}
```

```
public class Tester
{
  public static void myFunc(Point loc)
  {
    loc.X = 50;
    loc.Y = 100;
    Console.WriteLine(‘‘In MyFunc loc: {0}’’, loc);
  }
  static void Main( )
  {
    Point loc1 = new Point(200,300);
    Console.WriteLine(‘‘Loc1 location: {0}’’, loc1);
    myFunc(loc1);
    Console.WriteLine(‘‘Loc1 location: {0}’’, loc1);
  }
}
```

După cum este dat în exemplul de mai sus, crearea unei instanţe se face folosind operatorul *new*; dar în acest caz, nu se va crea o instanţă în memoria heap, ci pe stivă. Transmiterea lui loc1 ca parametru se face prin valoare, adică metoda *myFunc* nu face decât să modifice o copie de pe stivă a lui *loc1*. La revenire, se va afişa tot valoarea originală, deoarece *loc1* a rămas nemodificat:

```
Loc1 location: 200, 300
In MyFunc loc: 50, 100
Loc1 location: 200, 300
```

Deseori pentru o structură se declară câmpurile ca fiind publice, pentru a nu mai fi necesare definirea accesorilor (simplificare implementării). Alţi programatori consideră însă că accesarea membrilor trebuie să se facă precum la clase, folosind proprietăţi. Oricare ar fi alegerea, limbajul o sprijină.

Alte aspecte demne de reţinut:

- Câmpurile nu pot fi iniţializate la declarare; altfel spus, dacă în exemplul de mai sus se scria:

  ```
  public int xVal = 10;
  public int yVal = 20;
  ```

  s-ar fi semnalat o eroare la compilare.

- Nu se poate defini un constructor implicit. Cu toate acestea, compilatorul va crea un astfel de constructor, care va iniţializa câmpurile la valorile lor implicite (0 pentru tipuri numerice sau pentru enumerări, *false* pentru *bool*, *null* pentru tipuri referinţă).

  Pentru tipul *Point* de mai sus, următoarea secvenţă de cod este corectă:

  ```
  Point a = new Point(0, 0);
  Point b = new Point();
  ```

  şi duce la crearea a două puncte cu abcisele şi ordonatele 0. Un constructor implicit este apelat atunci când se creează un tablou de structuri:

  ```
  Point[] points = new Points[10];
  for( int i=0; i<points.Length; i++ )
  {
    Console.WriteLine(points[i]);
  }
  ```

  va afişa de 10 ori puncte de coordonate (0, 0). De asemenea este apelat la iniţializarea membrilor unei clase (care se face înainte de orice constructor).

  De menţionat pentru exemplul anterior că se creează un obiect de tip tablou în heap, după care în interiorul lui (şi nu pe stivă!) se creează cele 10 puncte (alocare inline).

- Nu se poate declara destructor. Acesţia se declară numai pentru clase.

- Dacă programatorul defineşte un constructor, atunci acesta trebuie să dea valori iniţiale pentru câmpurile conţinute, altfel apare eroare la compilare.

- Dacă pentru instanţierea unei structuri declarate în interiorul unei metode sau bloc de instrucţiuni în general nu se apelează *new*, atunci respectiva instanţă nu va avea asociată nici o valoare (constructorul implicit nu este apelat automat!). Nu se poate folosi respectiva variabilă de tip structură decât după ce i se iniţializează toate câmpurile:

  ```
  Point p;
  Console.WriteLine(p);
  ```

  va duce la apariţia erorii de compilare:

```
Use of unassigned local variable 'p'
```

Dar după nişte asignări de tipul:

```
p.xVal=p.yVal=0;
```

afişarea este posibilă (practic, orice apel de metodă pe instanţă este acum acceptat).

- Dacă se în cearcă definirea unei structuri care conţine un câmp de tipul structurii, atunci va apărea o eroare de compilare:

```
struct MyStruct
{
  MyStruct s;
}
```

va genera un mesaj din partea compilatorului:

```
Struct member 'MyStruct.s' of type 'MyStruct' causes a cycle in the stru
```

- Dacă o instanţă este folosită acolo unde un *object* este necesar, atunci se va face automat o conversie implicită către *System.Object* (boxing). Ca atare, utilizarea unei structuri poate duce (dar nu obligatoriu, ci în funcţie de context) la un overhead datorat conversiei.

6.3.1 Structuri sau clase?

Structurile pot fi mult mai eficiente în alocarea memoriei atunci când sunt reţinute într–un tablou. De exemplu, crearea unui tablou de 100 de elemente de tip *Point* (de mai sus) va duce la crearea unui singur obiect (tabloul este un obiect), iar cele 100 de instanţe de tip structură ar fi alocate inline în vectorul creat (şi nu referinţe ale acestora). Dacă *Point* ar fi declarat ca şi clasă, ar fi fost necesară crearea a 101 instanţe de obiecte în heap (un obiect pentru tablou, alte 100 pentru puncte), ceea ce ar duce la mai mult lucru pentru garbage collector.

Dar în cazul în care structurile sunt folosite în colecţii (care stochează *object*), se va face automat un boxing, ceea ce duce la overhead (memorie şi timp = resurse suplimentare). De asemenea, la transmiterea prin valoare a unei structuri, se va face copierea tuturor câmpurilor conţinute pe stivă, ceea ce poate duce la un overhead semnificativ.

Curs 7

Tratarea excepţiilor. Atribute

7.1 Tratarea excepţiilor

C#, la fel ca alte limbaje, permite tratarea erorilor şi a situaţiilor deosebite prin excepţii. O excepţie este un obiect care încapsulează informaţie despre o situaţie anormală. Ea este folosită pentru a semnala contextul în care apare situaţia deosebită

Un programator nu trebuie să confunde tratarea excepţiilor cu erorile sau bug–urile. Un bug este o eroare de programare care ar trebui să fie fixată înainte de livrarea codului. Excepţiile nu sunt gândite pentru a preveni bug–urile (cu toate că un bug poate să ducă la apariţia unei excepţii), pentru că acestea din urmă ar trebui să fie eliminate.

Chiar dacă se scot toate bug–urile, vor exista erori predictibile dar neprevenibile, precum deschiderea unui fişier al cărui nume este greşit sau împărţiri la 0. Nu se pot preveni astfel de situaţii, dar se pot manipula astfel încât nu vor duce la prăbuşirea programului. Când o metodă întâlneşte o situaţie excepţională, atunci se va arunca o excepţie; cineva va trebui să sesizeze (să "prindă") această excepţie, sau eventual să lase o funcţie de nivel superior să o trateze. Dacă nimeni nu tratează această excepţie, atunci CLR o va face, dar aceasta duce la oprirea *thread–ului*[1].

7.1.1 Tipul *Exception*

În C# se pot arunca ca excepţii obiecte de tip *System.Exception* sau derivate ale acestuia. Există o ierarhie de excepţii care se pot folosi, sau se pot crea propriile tipuri excepţie.

[1] Şi nu neapărat a întregului proces!

Enumerăm următoarele metode şi proprietăţi relevante ale clasei *Exception*:

- *public Exception(), public Exception(string), public Exception(string, Exception)* - constructori; ultimul preia un obiect de tip *Exception* (sau de tip clasă derivată) care va fi încapsulat în instanţa curentă; o excepţie poate deci să conţină în interiorul său o instanţă a altei excepţii (cea care a fost de fapt semnalată iniţial).
- *public virtual string HelpLink* {*get; set;*} obţine sau setează un link către un fişier help asociat acestei excepţii; poate fi de asemenea o adresa Web (URL)
- *public Exception InnerException* {*get;*} returnează excepţia care este încorporată în excepţia curentă
- *public virtual string Message* {*get;*} obţine un mesaj care descrie excepţia curentă
- *public virtual string Source* {*get; set;*} obţine sau setează numele aplicaţiei sau al obiectului care a cauzat eroarea
- *public virtual string StackTrace* {*get;*} obţine o reprezetare string a apelurilor de metode care au dus la apariţia acestei excepţii
- *public MethodBase TargetSite* {*get;*} obţine metoda care a aruncat excepţia curentă[2]

7.1.2 Aruncarea şi prinderea excepţiilor

Aruncarea cu *throw*

Aruncarea unei excepţii se face folosind instrucţiunea *throw*. Exemplu:

```
throw new System.Exception();
```

Aruncarea unei excepţii opreşte execuţia metodei curente, după care CLR începe să caute un manipulator de excepţie. Dacă un handler de excepţie nu este găsit în metoda curentă, atunci CLR va curăţa stiva, ajungându–se la metoda apelantă. Fie undeva în lanţul de metode care au fost apelate se găseşte un exception handler, fie thread–ul curent este terminat de către CLR.

Exemplu:

[2]MethodBase este o clasă care pune la dispoziţie informaţii despre metodele şi constructorii unei clase

```
using System;
public class Test
{
  public static void Main( )
  {
    Console.WriteLine(‘‘Enter Main...’’);
    Test t = new Test( );
    t.Func1( );
    Console.WriteLine(‘‘Exit Main...’’);
  }
  public void Func1( )
  {
    Console.WriteLine(‘‘Enter Func1...’’);
    Func2( );
    Console.WriteLine(‘‘Exit Func1...’’);
  }
  public void Func2( )
  {
    Console.WriteLine(‘‘Enter Func2...’’);
    throw new System.Exception( );
    Console.WriteLine(‘‘Exit Func2...’’);
  }
}
```

Se exemplifică apelul de metode: *Main()* apelează *Func1()*, care apelează *Func2()*; aceasta va arunca o excepţie. Deoarece lipseşte un event handler care să trateze această excepcţie, se va întrerupe thread–ul curent (şi fiind singurul, şi întregul proces) de către CLR, iar la ieşire vom avea:

```
Enter Main...
Enter Func1...
Enter Func2...
Exception occurred: System.Exception: An exception of type
System.Exception was thrown at Test.Func2( )
in ...Test.cs:line 24
at Test.Func1( )
in ...Test.cs:line 18
at Test.Main( )
in ...Test.cs:line 12
```

Deoarece este aruncată o excepţie, în metoda *Func2()* nu se va mai executa ultima linie, ci CLR–ul va începe imediat căutarea event handler–ului care

să trateze excepţia. La fel, nu se execută nici ultima linie din *Func1()* sau din *Main()*.

Prinderea cu *catch*

Prinderea şi tratarea excepţiei se poate face folosind un bloc *catch*, creat prin intermediul instrucţiunii *catch*.

Exemplu:

```
using System;
public class Test
{
  public static void Main( )
  {
    Console.WriteLine(‘‘Enter Main...’’);
    Test t = new Test( );
    t.Func1( );
    Console.WriteLine(‘‘Exit Main...’’);
  }
  public void Func1( )
  {
    Console.WriteLine(‘‘Enter Func1...’’);
    Func2( );
    Console.WriteLine(‘‘Exit Func1...’’);
  }
  public void Func2( )
  {
    Console.WriteLine(‘‘Enter Func2...’’);
    try
    {
      Console.WriteLine(‘‘Entering try block...’’);
      throw new System.Exception( );
      Console.WriteLine(‘‘Exiting try block...’’);
    }
    catch
    {
      Console.WriteLine(‘‘Exception caught and handled.’’);
    }
    Console.WriteLine(‘‘Exit Func2...’’);
  }
}
```

Se observă că s–a folosit un bloc *try* pentru a delimita instrucţiunile care vor duce la apariţia excepţiei. În momentul în care se aruncă excepţia, restul instrucţiunilor din blocul *try* se ignoră şi controlul este preluat de către blocul *catch*. Deoarece excepţia a fost tratată, CLR–ul nu va mai opri procesul. La ieşire se va afişa:

```
Enter Main...
Enter Func1...
Enter Func2...
Entering try block...
Exception caught and handled.
Exit Func2...
Exit Func1...
Exit Main...
```

Se observă că în blocul *catch* nu s–a specificat tipul de excepţie care se prinde; asta înseamnă că se va prinde orice excepţie se va arunca, indiferent de tipul ei. Chiar dacă excepţia este tratată, execuţia nu se va relua de la instrucţiunea care a produs excepţia, ci se continuă cu instrucţiunea de după blocul *catch*.

Uneori, prinderea şi tratatarea excepţiei nu se poate face în funcţia apelată, ci doar în funcţia apelantă. Exemplu:

```
using System;
public class Test
{
  public static void Main( )
  {
    Console.WriteLine(‘‘Enter Main...’’);
    Test t = new Test( );
    t.Func1( );
    Console.WriteLine(‘‘Exit Main...’’);
  }
  public void Func1( )
  {
    Console.WriteLine(‘‘Enter Func1...’’);
    try
    {
      Console.WriteLine(‘‘Entering try block...’’);
      Func2( );
      Console.WriteLine(‘‘Exiting try block...’’);
    }
```

```
    catch
    {
      Console.WriteLine(‘‘Exception caught and handled.’’);
    }
    Console.WriteLine(‘‘Exit Func1...’’);
  }
  public void Func2( )
  {
    Console.WriteLine(‘‘Enter Func2...’’);
    throw new System.Exception( );
    Console.WriteLine(‘‘Exit Func2...’’);
  }
}
```

La ieşire se va afişa:

```
Enter Main...
Enter Func1...
Entering try block...
Enter Func2...
Exception caught and handled.
Exit Func1...
Exit Main...
```

Este posibil ca într–o secvenţă de instrucţiuni să se arunce mai multe tipuri de excepţii, în funcţie de natura stării apărute. În acest caz, prinderea excepţiei printr–un bloc *catch* generic, ca mai sus, nu este utilă; am vrea ca în funcţie de natura excepţiei aruncate, să facem o tratare anume. Se sugereaza chiar să nu se folosească această construcţie de prindere generică, deoarece în majoritatea cazurilor este necesar să se cunoască natura erorii (de exemplu pentru a fi scrisă într-un fişier de logging, pentru a fi consultată mai târziu).

Acest lucru se face specificând tipul excepţiei care ar trebui tratate în blocul *catch*:

```
using System;
public class Test
{
  public static void Main( )
  {
    Test t = new Test( );
    t.TestFunc( );
  }
```

```
  //incearca sa imparta doua numere
  public void TestFunc( )
  {
    try
    {
      double a = 5;
      double b = 0;
      Console.WriteLine (``{0} / {1} = {2}'', a, b, DoDivide(a,b));
    }
    //cel mai derivat tip de exceptie se specifica primul
    catch (System.DivideByZeroException)
    {
      Console.WriteLine(``DivideByZeroException caught!'');
    }
    catch (System.ArithmeticException)
    {
      Console.WriteLine(``ArithmeticException caught!'');
    }
    //Tipul mai general de exceptie este ultimul
    catch
    {
      Console.WriteLine(``Unknown exception caught'');
    }
  }
  //efectueaza impartirea daca se poate
  public double DoDivide(double a, double b)
  {
    if (b == 0)
      throw new System.DivideByZeroException( );
    if (a == 0)
      throw new System.ArithmeticException( );
    return a/b;
  }
}
```

În exemplul de mai sus s–a convenit ca o împărţire cu numitor 0 să ducă la o execepţie *System.DivideByZeroException*, iar o împărţire cu numărător 0 să ducă la apariţia unei excepţii de tip *System.ArithmeticException*. Este posibilă specificarea mai multor blocuri de tratare a excepţiilor. Aceste blocuri sunt parcurse în ordinea în care sunt specificate, iar primul tip care se potriveşte cu excepţia aruncată (în sensul că tipul excepţie specificat este

fie exact tipul obiectului aruncat, fie un tip de bază al acestuia - din cauză de upcasting) este cel care va face tratarea excepţiei apărute. Ca atare, este important ca ordinea excepţiilor tratate să fie de la cel mai derivat la cel mai general. În exemplul anterior, *System.DivideByZeroException* este derivat din clasa *System.ArithmeticException*.

Blocul *finally*

Uneori, aruncarea unei excepţii şi golirea stivei până la blocul de tratare a excepţiei poate să nu fie o idee bună. De exemplu, dacă excepţia apare atunci când un fişier este deschis (şi închiderea lui se poate face doar în metoda curentă), atunci ar fi util ca să se închidă fişierul înainte ca să fie preluat controlul de către metoda apelantă. Altfel spus, ar trebui să existe o garanţie că un anumit cod se va executa, indiferent dacă totul merge normal sau apare o excepţie. Acest lucru se face prin intermediul blocului *finally*, care se va executa în orice situaţie. Existenţa acestui bloc elimină necesitatea existenţei blocurilor *catch* (cu toate că şi acestea pot să apară).

Exemplu:

```
using System;
public class Test
{
  public static void Main( )
  {
    Test t = new Test( );
    t.TestFunc( );
  }
  public void TestFunc( )
  {
    try
    {
      Console.WriteLine(‘‘Open file here’’);
      double a = 5;
      double b = 0;
      Console.WriteLine (‘‘{0} / {1} = {2}’’, a, b, DoDivide(a,b));
      Console.WriteLine (‘‘This line may or may not print’’);
    }
    finally
    {
      Console.WriteLine (‘‘Close file here.’’);
    }
```

```
  }
  public double DoDivide(double a, double b)
  {
    if (b == 0)
    throw new System.DivideByZeroException( );
    if (a == 0)
    throw new System.ArithmeticException( );
    return a/b;
  }
}
```

În exemplul de mai sus, mesajul "Close file here" se va afişa indiferent de ce parametri se transmit metodei *DoDivide()*.

La aruncarea unei excepţii se poate particulariza obiectul care se aruncă:

```
if (b == 0)
{
  DivideByZeroException e = new DivideByZeroException( );
  e.HelpLink = ''http://www.greselifatale.com'';
  throw e;
}
```

iar când excepţia este prinsă, se poate prelucra informaţia:

```
catch (System.DivideByZeroException e)
{
  Console.WriteLine( ''DivideByZeroException!
    goto {0} and read more'', e.HelpLink);
}
```

Crearea propriilor excepţii

În cazul în care suita de excepţii predefinite nu este suficientă, programatorul îşi poate construi propriile tipuri. Se recomandă ca acestea să fie derivate din *System.ApplicationException*, care este derivată direct din *System.Exception*. Se indică această derivare deoarece astfel se face distincţie între excepţiile aplicaţie şi cele sistem (cele aruncate de către CLR).

Exemplu:

```
using System;
public class MyCustomException : System.ApplicationException
{
  public MyCustomException(string message): base(message)
```

```
  {
  }
}

public class Test
{
  public static void Main( )
  {
    Test t = new Test( );
    t.TestFunc( );
  }

  public void TestFunc( )
  {
    try
    {
      double a = 0;
      double b = 5;
      Console.WriteLine (``{0} / {1} = {2}'', a, b, DoDivide(a,b));
      Console.WriteLine (``This line may or may not print'');
    }
    catch (System.DivideByZeroException e)
    {
      Console.WriteLine(``DivideByZeroException! Msg: {0}'',
        e.Message);
      Console.WriteLine(``HelpLink: {0}'',
        e.HelpLink);
    }
    catch (MyCustomException e)
    {
      Console.WriteLine(``\nMyCustomException! Msg: {0}'',
        e.Message);
      Console.WriteLine(``\nHelpLink: {0}\n'',
        e.HelpLink);
    }
    catch
    {
      Console.WriteLine(``Unknown exception caught'');
    }
  }
```

```
  public double DoDivide(double a, double b)
  {
    if (b == 0)
    {
      DivideByZeroException e = new DivideByZeroException( );
      e.HelpLink= ‘‘http://www.greselifatale.com’’;
      throw e;
    }
    if (a == 0)
    {
      MyCustomException e = new MyCustomException( ‘‘Can’t have
        zero divisor’’);
      e.HelpLink = ‘‘http://www.greselifatale.com/NoZeroDivisor.htm’’;
      throw e;
    }
    return a/b;
  }
}
```

Rearuncarea excepţiilor

Este perfect posibil ca într–un bloc de tratare a excepţiilor să se se facă o tratare primară a excepţiei, după care să se arunce mai departe o altă excepţie, de acelaşi tip sau de tip diferit (sau chiar excepţia originală). Dacă se doreşte ca această excepţie să păstreze cumva în interiorul ei excepţia originală, atunci constructorul permite înglobarea unei referinţe la aceasta; această referinţă va fi accesibilă prin intermediul proprietăţii *InnerException*:

```
using System;
public class MyCustomException : System.ApplicationException
{
  public MyCustomException(string message,Exception inner):
    base(message,inner)
  {
  }
}
public class Test
{
  public static void Main( )
  {
    Test t = new Test( );
```

```
    t.TestFunc( );
  }
  public void TestFunc( )
  {
    try
    {
      DangerousFunc1( );
    }
    catch (MyCustomException e)
    {
      Console.WriteLine(``\n{0}'', e.Message);
      Console.WriteLine(``Retrieving exception history...'');
      Exception inner = e.InnerException;
      while (inner != null)
      {
        Console.WriteLine(``{0}'',inner.Message);
        inner = inner.InnerException;
      }
    }
  }
  public void DangerousFunc1( )
  {
    try
    {
      DangerousFunc2( );
    }
    catch(System.Exception e)
    {
      MyCustomException ex = new MyCustomException(``E3 -
        Custom Exception Situation!'',e);
      throw ex;
    }
  }
  public void DangerousFunc2( )
  {
    try
    {
      DangerousFunc3( );
    }
    catch (System.DivideByZeroException e)
    {
```

```
      Exception ex =
      new Exception(``E2 - Func2 caught divide by zero'',e);
      throw ex;
    }
  }
  public void DangerousFunc3( )
  {
    try
    {
      DangerousFunc4( );
    }
    catch (System.ArithmeticException)
    {
      throw;
    }
    catch (System.Exception)
    {
      Console.WriteLine(``Exception handled here.'');
    }
  }
  public void DangerousFunc4( )
  {
    throw new DivideByZeroException("E1 - DivideByZero Exception");
  }
}
```

7.1.3 Reîncercarea codului

Se poate pune întrebarea: cum se procedează dacă se dorecşte revenirea la codul care a produs excepţia, după tratarea ei? Există destule situaţii în care reexecutarea acestui cod este dorită: să ne gândim de exemplu la cazul în care într-o fereastră de dialog se specifică numele unuei fişier ce trebuie procesat, numele este introdus greşit si se doreşte ca să se permită corectarea numelui. Un alt exemplu clasic este cazul în care autorul unei metode ştie că o operaţie poate să eşueze periodic – de exemplu din cauza unui timeout pe reţea – dar vrea să reîncerce operaţia de n ori înainte de a semnala eroare.

În această situaţie se poate defini o etichetă înaintea blocului `try` la care să se permită salutl printr–un `goto`. Următorul exemplu permite unui utilizator să specifice de maxim trei ori numele unui fişier ce se procesează, cu revenire în cazul erorii.

```
using System;
using System.IO;

class Retry
{

    static void Main()
    {
        StreamReader sr;

        int attempts = 0;
        int maxAttempts = 3;

        GetFile:
        Console.Write("\n[Attempt #{0}] Specify file " +
            "to open/read: ", attempts+1);
        string fileName = Console.ReadLine();

        try
        {
            sr = new StreamReader(fileName);

            Console.WriteLine();

            string s;
            while (null != (s = sr.ReadLine()))
            {
                Console.WriteLine(s);
            }
            sr.Close();
        }
        catch(FileNotFoundException e)
        {
            Console.WriteLine(e.Message);
            if (++attempts < maxAttempts)
            {
                Console.Write("Do you want to select " +
                    "another file: ");
                string response = Console.ReadLine();
                response = response.ToUpper();
                if (response == "Y") goto GetFile;
```

```
            }
            else
            {
                Console.Write("You have exceeded the maximum " +
                    "retry limit ({0})", maxAttempts);
            }


        }
        catch(Exception e)
        {
            Console.WriteLine(e.Message);
        }

        Console.ReadLine();
}
}
```

7.1.4 Compararea tehnicilor de manipulare a erorilor

Metoda standard de tratare a aerorilor a fost în general returnarea unui cod de eroare către metoda apelantă. Ca atare, apelantul are sarcina de a descifra acest cod de eroare şi să reacţioneze în consecinţă. Însă aşa cum se arată mai jos, tratarea excepţiilor este superioară acestei tehnici din mai multe motive.

Neconsiderarea codului de retur

Apelul unei funcţii care returnează un cod de eroare poate fi făcut şi fără a utiliza efectiv codul returnat, scriind doar numele funcţiei cu parametrii de apel. Dacă de exemplu pentru o anmită procesare se apelează metoda `A` (de exemplu o deschidere de fişier) după care metoda `B` (citirea din fişier), se poate ca în `A` să apară o eroare care nu este luată în considerare; apelul lui `B` este deja sortit eşecului, pentru că buna sa funcţionare depinde de efectele lui `A`. Dacă însă metoda `A` aruncă o excepţie, atunci nici măcar nu se mai ajunge la apel de `B`, deoarece CLR-ul va pasa execuţia unui bloc `catch/finally`. Altfel spus, nu se permite o propagare a erorilor.

Manipularea erorii în contextul adecvat

În cazul în care o metodă A apelează alte metode $B_1, \ldots B_n$, este posibil ca oricare din aceste n metode să cauzeze o eroare (şi să returneze cod adecvat); tratarea erorii din exteriorul lui A este dificilă în acest caz, deoarece ar trebui să se cerceteze toate codurile de eroare posibile pentru a determina motivul apariţiei erorii. Dacă se mai adauga şi apelul de metodă B_{n+1} în interiorul lui A, atunci orice apel al lui A trebuie să includă suplimentar şi verificarea pentru posibilitatea ca B_{n+1} să fi cauzat o eroare. Ca atare, costul menţinerii codului creşte permanent, ceea ce are un impact negativ asupra TCO-ului[3]

Folosind tratarea excepţiilor, aruncând excepţii cu mesaje de eroare explicite sau excepţii de un anumit tip (definit de programator) se poate trata mult mai convenabil o situaţie deosebită. Mai mult decât atât, introducerea apelului lui B_{n+1} în interiorul lui A nu reclamă modificare suplimentară, deoarece tipul de excepţie aruncat de B_{n+1} este deja tratat (desigur, se presupune că se defineşte un tip excepţie sau o ierarhie de excepţii creată convenabil).

Uşurinţa citirii codului

Pentru comparaţie, se poate scrie un cod care realizează procesarea conţinutului unui fişier folosind coduri de eroare returnate sau excepţii. În primul caz, soluţia va conţine cod de prelucrare a conţinutului mixat cu cod de verificare şi reacţie pentru diferitele cazuri de excepţie. Codul în al doilea caz este mult mai scurt, mai uşor de înţeles, de menţinut, de corectat şi extins.

Aruncarea de excepţii din constructori

Nimic nu opreşte ca o situaţie deosebită să apară într–un apel de constructor. Tehnica verificării codului de retur nu mai funcţionează aici, deoarece un constructor nu returnează valori. Folosirea excepţiilor este în acest caz aproape de neînlocuit.

7.1.5 Sugestie pentru lucrul cu excepţiile

În Java, programatorii trebuei să declare că o metodă poate arunca o excepţie şi să o declare explicti într–o listă astfel încât un apelant să ştie că se poate se poate aştepta la primirea ei. Această cunoaştere în avans permite conceperea unui plan de lucru cu fiecare dintre ele, preferabil decât să se prindă oricare dintre ele cu un *catch* generic. În cazul .NET se sugerează

[3]Total Cost of Ownership.

să se menţină o documentaţie cu excepţiile care poti fi aruncate de fiecare metodă.

7.2 Atribute

O aplicaţie .NET conţine cod, date şi *metadate*. Metadata este o dată despre assembly (versiune, producator, etc), tipuri de date conţinute, etc stocate împreună cu codul compilat.

Atributele sunt folosite pentru a da o extra-informaţie compilatorului de .NET. Java foloseşte o combinaţie de semne /** şi @ pentru a include informaţie relativ la clase, metode, câmpuri sau parametri. Aceste comentarii însă nu vor fi incluse în bytecod-ul final, doar în eventuala documentaţie. Folosind atributele, această informaţie poate fi stocată în codul compilat şi reutilizată ulterior la runtime.

Unde ar fi utile aceste atribute? Exemple de utilizare a lor ar fi urmărirea bug–urilor care există într–un sistem sau urmărirea stadiului proiectului. De asemenea, anumite atribute predefinite sunt utilizate pentru a specifica dacă un tip este sau nu serializabil, care sunt porţiunile de cod scoase din circulaţie, informaţii despre versiunea assembly-ului, etc.

7.2.1 Generalităţi

Atributele sunt de două feluri: intrinseci (predefinite) şi definite de utilizator. Cele intrinseci sunt integrate în platforma .NET şi sunt recunoscute de CLR. Atributele definite de utilizator sunt create în funcţie de dorinţele acestuia.

Atributele se pot specifica pentru: assembly–uri, clase, constructori, delegaţi, enumerări, evenimente, câmpuri, interfeţe, metode, module, parametri, proprietăţi, valori de retur, structuri.

Ţinta unui atribut specifică cui anume i se va aplica un atribut anume. Tabelul 7.1 conţine ţintele posibile şi o scurtă descriere a lor.

Tabelul 7.1: Ţintele atributelor.

Ţintă	Descriere
All	Orice element
Assembly	Un assembly
ClassMembers	Orice membru al unei clase
Class	O clasă
Constructor	Un constructor

Tabelul 7.1 (continuare)

Ţintă	Descriere
Delegate	Un delegat
Enum	O enumerare
Event	Un eveniment
Field	Un câmp
Interface	O interfaţă
Method	O metodă
Module	Un modul
Parameter	Un parametru
Property	O proprietate
ReturnValue	O valoare de retur
Struct	O structură

Aplicarea atributelor se face prin specificarea numelui lor între paranteze drepte; mai multe atribute fie se specificăunul deasupra celuilalt, fie în interiorul aceloraşi paranteze, despărţite prin virgulă. Exemplu:

```
[Serializable]
[Webservice]
```

echivalent cu:

```
[Serializable, Webservice]
```

În cazul atributelor ce se specifică pentru un assembly, forma lor este:

```
[assembly:AssemblyDelaySign(false)]
```

în cazul acestor din urmă atribute, specificarea lor se face după toate declaraţiile `using`, dar înaintea oricărui cod. În general, specificarea unui atribut se face prin scrierea lui imediat înaintea elementului asupra căruia se aplică:

```
using System;
[Serializable]
class ClasaSerializabila
{
//definitia clasei
}
```

7.2.2 Atribute predefinite

Tabelul 7.2 prezintă câteva dintre ele:

Tabelul 7.2: Atribute predefinite.

Atribut	Descriere
System.SerializableAttribute [Serializable]	Permite unei clase să fie serializată pe disc sau într–o reţea
System.NonSerializedAttribute [NonSerialized]	Permite unor membri să nu fie salvaţi pe reţea sau pe disc
System.Web.Services.WebServiceAttribute [WebService]	Permite specificarea unui nume şi a unei descrieri pentru un serviciu Web
System.Web.Services.WebMethodAttribute [WebMethod]	Marchează o metodă ca fiind expusă ca parte a unui serviciu Web
System.AttributeUsageAttribute [AttributeUsage]	Defineşte parametrii de utilizare pentru atribute
System.ObsoleteAttribute [Obsolete]	Marchează o secvenţă ca fiind scoasă din uz
System.Reflection.AssemblyVersionAttribute [AssemblyVersion]	Specifică numărul de versiune al unui assembly
System.Attribute.CLSCompliant [CLSCompliant]	Indică dacă un element de program este compatibil cu CLS
System.Runtime.InteropServices. DllImportAttribute [DllImport]	specifică locaţia DLL care conţine implementarea pentruo metodă externă

Între parantezele drepte se arată cum se specifică acel atribut.

Exemplul de mai jos foloseşte atributul *System.ObsoleteAttribute*.

```
using System;

namespace AttributeSample1
{
class Class1
{
[STAThread]
static void Main(string[] args)
{
int s1 = AddTwoNumbers(2, 3);
int s2 = AddNumbers(2, 3);
int s3 = AddNumbers(2, 3, 4);
}

[Obsolete("obsolete: use AddNumbers instead")]
```

```
static int AddTwoNumbers(int a, int b)
{
return a + b;
}

static int AddNumbers(params int[] numbers)
{
int result = 0;
foreach(int number in numbers)
result += number;
return result;
}
}
}
```

La compilarea codului se generează un avertisment care semnalizează faptul că se utilizează o metodă care este scoasă din uz. Mesajul specificat ca parametru al atributului este afişat ca mesaj al avertismentului. Suplimentar, pentru atributul Obsolete se poate specifica dacă respectivul avertisment este sau nu interpretat ca o eroare, adăugând un parametru de tip bool: *false* ca la compilare să se genereze avertisment, *true* pentru ca utilizarea să fie tratată ca o eroare.

Exemplul de mai jos exemplifică serializarea şi deserializarea unui obiect:

```
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
[Serializable]
class Point2D
{
  public int X;
  public int Y;
}
class MyMainClass
{
  public static void Main()
  {
    Point2D My2DPoint = new Point2D();
    My2DPoint.X = 100;
    My2DPoint.Y = 200;
    Stream WriteStream = File.Create("Point2D.bin");
    BinaryFormatter BinaryWrite = new BinaryFormatter();
```

```
    BinaryWrite.Serialize(WriteStream, My2DPoint);
    WriteStream.Close();
    Point2D ANewPoint = new Point2D();
    Console.WriteLine("New Point Before Deserialization: ({0}, {1})",
    ANewPoint.X, ANewPoint.Y);
    Stream ReadStream = File.OpenRead("Point2D.bin");
    BinaryFormatter BinaryRead = new BinaryFormatter();
    ANewPoint = (Point2D)BinaryRead.Deserialize(ReadStream);
    ReadStream.Close();
    Console.WriteLine("New Point After Deserialization: ({0}, {1})",
    ANewPoint.X, ANewPoint.Y);
  }
}
```

7.2.3 Exemplificarea altor atribute predefinite

Atributul *Conditional*

Acest atribut se ataşează la o metodă pentru care se doreşte ca atunci când compilatorul o întâlneşte la un apel, dacă un anumit simbol nu e definit atunci nu va fi chemată. Este folosită pentru a omite anumite apeluri de metode (de exemplu cele folosite la etapa de debugging).

Exemplu:

```
using System;
using System.Diagnostics;

namespace CondAttrib
{
  class Thing
  {
    private string name;
    public Thing(string name)
    {
      this.name = name;
      SomeDebugFunc();
      SomeFunc();
    }
    public void SomeFunc()
    { Console.WriteLine("SomeFunc"); }

    [Conditional("DEBUG")]
```

```
    public void SomeDebugFunc()
    { Console.WriteLine("SomeDebugFunc"); }
  }

  public class Class1
  {
    [STAThread]
    static void Main(string[] args)
    {
      Thing t = new Thing("T1");
    }
  }
}
```

Definirea unui anumit simbol (de exemplu DEBUG) se poate face în două moduri:

- prin folosirea unei directive de preprocesor de tipul #define:

  ```
  #define DEBUG
  ```

- prin folosirea parametrilor din linia de comanda sau a posibilităţilor mediului integrat de dezvoltare. De exemplu, pentru a se defini un anumit simbolul din Visual Studio se procedează astfel: clic dreapta în Solution Explorer pe numele proiectului, selectare Properties apoi Configuration Properties. Pe linia Conditional Compilation Constants se adaugă simbolul dorit. Acest lucru va avea ca efect folosirea opţiunii *define* a compilatorului.

Dacă la compilare simbolul nu este definit atunci compilatorul va ignora apelurile de metode calificate cu atributul Conditional. Acest lucru se paote verifica atât urmărind execuţia programului, cât şi din inspecţia codului IL rezultat la compilare.

Atributul CLSCompliant

Acest atribut se aplică pe assembly–uri. Dacă un assembly este marcat ca fiind *CLSCompliant*, orice tip expus public în assembly care nu este compatibil CLS trebuie să fie marcat cu *CLSCompliant(false)*. De exmplu, CLS prevede faptul că tipurile de date întregi ar trebui să fie cu semn. O clasă poate să conţină membri (câmpuri, metode) de tip unsigned, dar dacă respectiva clasă este declarată ca fiind CLSCompliant atunci acestea ar trebui

declarate ca fiind ne-vizibile din afara assembly–ului. Dacă ele sunt expuse în afara assembly–ului, unele limbaje .NET s-ar putea să nu le poată folosi efetiv. Pentru a ajuta dezvoltatorii .NET să evite asemenea situaţii, platforma pune la dispoziţie atributul CLSCompliant cu care se controlează răspunsul compilatorului la expunerea entităţilor ne-compatibile cu CLS; astfel, se va genera o eroare sau se vor ignora asemenea cazuri.

Exemplu:

```
[assembly:CLSCompliant(true)]
namespace DemoCLS
{
public class ComplianceTest
{
    //Tipul uint nu ste compatibil CLS
    //deaorece acest camp este privat, regulile CLS nu se aplica
    private uint a = 4;

    //deoarece acest camp uint este public, avem
    //incompatibilitate CLS
    public uint B = 5;

    //Acesta este modul corect de expunere a unui uint:
    //tipul long este compatibil CLS
    public long A
    {
       get { return a; }
    }
}
}
```

Dacă se compilează assembly-ul de mai sus atunci apare o eroare:

```
Type of 'DemoCLS.ComplianceTest.B' is not CLS-compliant
```

Pentru ca această eroare să nu mai apară putem să declarăm B ca fiind invizibil din exteriorul assembly-ului sau să adăugăm inaintea lui B atributul:

```
[CLSCompliant(false)]
```

Meritul acestui atribut este că anunţă programatorul despre eventualele neconcordanţe cu Common Language Specifications.

7.2.4 Atribute definite de utilizator

În general, atributele predefinite acoperă marea majoritate a situaţiilor care cer utilizarea de atribute. Eventualizatea ca utilizatorul să dorească crearea propriilor sale atribute este prevăzută de către platforma .NET, dându–se posibilitatea definirii lor.

Există câteva situaţii în care e benefică definirea de noi atribute. Exemplul cel mai des întâlnit este acela în care pentru a se menţine informaţii despre un cod la care lucrează mai multe echipe, se defineşte un atribut care să servească la pasarea de informaţie relativ la porţiuni din cod. Un alt exemplu este utilizarea unui sistem de urmărire a stadiului de dezvoltare a codului, care ar folosi informaţia stocată în atribute.

Un atribut utilizator este o clasă definită ca până acum. Se cere a fi derivată din *System.Attribute* fie direct, fie indirect. Sunt câţiva paşi care trebuie parcurşi pentru realizara unui atribut: specificarea ţintei, derivarea ei adecvată, denumirea ei în conformitate cu anumite reguli (recomandat, dar nu obligatoriu), definirea clasei.

Pasul 1. Declararea ţintei

Primul pas în crearea unui atribut utilizator este specificarea domeniului său de aplicabilitate, adică a elementelor cărora li se poate ataşa. Ţintele sunt cele din tabelul 7.1, care sunt de fapt valori din enumerarea *AttributeTargets*. Specificarea ţintei se face prin intermediul unui (meta)atribut *AttributeUsage*. Pe lângă ţinta propriu–zisă, se mai pot specifica valorile proprietăţilor *Inherited* şi *AllowMultiple*.

Proprietatea *Inherited* este de tip boolean şi precizează dacă atributul poate fi moştenit de către clasele derivate din cele căreia i se aplică. Valoarea implicită este *true*.

Proprietatea *AllowMultiple* este de asemenea de tip boolean şi specifică dacă se pot utiliza mai multe instanţe ale atributului pe un acelaşi element. Valoarea implicită este *false*.

Vom defini mai jos ţinta atributului pe care îl vom construi ca fiind assembly–ul, clasa, metoda, cu posibilitate de repetare a sa:

```
[AttributeUsage(AttributeTargets.Assembly|AttributeTargets.Class
      | AttributeTargets.Method, AllowMultiple=true)]
```

Pasul 2. Declararea unei clase atribut

Pentru declararea unei clase atribut, următoarele reguli trebuie să fie avute în vedere:

- (obligatoriu) O clasă atribut trebuie să deriveze direct sau indirect *System.Attribute*
- (obligatoriu) O clasă atribut trebuie să fie declarată ca fiind publică
- (recomandat) Un atribut ar trebui să aibă sufixul *Attribute*.

Vom declara clasa atribut *CodeTrackerAttribute*. Acest atribut s-ar putea folosi în cazul în care s-ar dori urmărirea dezvoltării codului în cadrul unui proiect de dimensiuni foarte mari, la care participă mai multe echipe. Acest atribut utilizator va fi inclus în codul compilat şi distribuit altor echipe (care nu vor avea acces la cod). O alternativă ar fi folosirea documentatţiei XML generate după comentariile din cod.

```
[AttributeUsage(AttributeTargets.Assembly|AttributeTargets.Class
       | AttributeTargets.Method, AllowMultiple=true)]
public class CodeTrackerAttribute : System.Attribute
{
//aici e codul dumneavoastra
}
```

Pasul 3. Declararea constructorilor şi a proprietăţilor

Acest pas constă în definirea efectivă membrilor clasei. Vom considera cătributul pe care îl creăm va purta trei informaţii: numele programatorului care a acţionat asupra respectivei unităţi de cod, faza în care se află, opţional note. Primele două atribute sunt mandatorii şi vor fi preluate prin constructor, al treilea poate fi setat prin intermediul unei proprietăţi.

```
using System;
[AttributeUsage(AttributeTargets.Assembly|AttributeTargets.Class
       | AttributeTargets.Method, AllowMultiple=true)]
public class CodeTrackerAttribute : System.Attribute
{
private string name;
private string phase;
private string notes;

public CodeTrackerAttribute(string name, string phase)
{
  this.name = name;
  this.phase = phase;
}
```

```
public virtual string Name
{
  get{return name;}
}

public virtual string Phase
{
  get{return phase;}
}

public virtual string Notes
{
  get{return notes;}
  set{notes=value;}
}
}
```

Pasul 4. Utilizarea atributului utilizator

Atributele definite de utilizator sutn folosite în acelaşi mod ca şi cele implicite. Codul de mai jso exemplifică acest lucru:

```
[CodeTracker("Lucian Sasu", "implementing specification",
  Notes = "First attempt")]
class AttribTest
{
  public AttribTest()
  {
    Console.WriteLine("AttribTest instance");
  }
  [CodeTracker("Lucian Sasu", "April 1st 2005")]
  public void SayHello(String message)
  {
    Console.WriteLine(message);
  }
}
```

O inspeţie a codului IL rezultat arată că aceste atribute s-au salvat în codul compilat. Obţinerea acestor atribute şi a valorilor lor se poate face foarte uşor prin reflectare.

Curs 8

ADO.NET

8.1 Ce reprezintă ADO.NET?

ADO.NET reprezintă o parte componentă a lui .NET Framework ce permite aducerea, manipularea şi modificarea datelor. În mod normal, o sursă de date poate să fie o bază de date, dar de asemenea un fişier text sau Excel sau XML sau Access. Lucrul se poate face fie conectat, fie deconectat de la sursa de date. Deşi există variate modalităţi de lucru cu bazele de date, ADO.NET se impune tocmai prin faptul că permite lucrul deconectat de la baza de date, integrarea cu XML, reprezentarea comună a datelor cu posibilitatea de a combina date din variate surse, toate pe baza unor clase .NET.

Faptul că se permite lucrul deconectat de la sursa de date rezolvă următoarele probleme:

- menţinerea conexiunilor la baza de date este o operaţie costisitoare. O bună parte a lăţimii de bandă este menţinută ocupată pentru nişte procesări care nu necesită neapărat conectare continuă
- probleme legate de scalabilitatea aplicaţiei: se poate ca serverul de baze de date să lucreze uşor cu 5-10 conexiuni menţinute, dar dacă numărul acestora creşte aplicaţia poate să reacţioneze extrem de lent
- pentru unele servere se impun clauze asupra numărului de conexiuni ce se pot folosi simultan.

Toate acestea fac ca ADO.NET să fie o tehnologie mai potrivită pentru dezvoltarea aplicaţiilor Internet decât cele precedente (e.g. ADO, ODBC).

Pentru o prezentare a metodelor de lucru cu surse de date sub platformă Windows se poate consulta [7].

Vom exemplifica în cele ce urmează preponderent pe bază de date Microsoft SQL 2000. Pentru lucrul cu MS SQL Server se poate instala gratuit versiunea gratuită de Microsoft SQL Server 2000 Desktop Engine[1].

8.2 Furnizori de date în ADO.NET

Din cauza existenţei mai multor tipuri de surse de date (de exemplu, a mai multor producători de servere de baze de date) e nevoie ca pentru fiecare tip major de protocol de comunicare să se folosească o bibliotecă specializată de clase. Toate aceste clase implementează nişte interfeţe bine stabilite, ca atare trecerea de la un SGBD la altul se face cu eforturi minore (dacă codul este scris ţinând cont de principiile POO).

Există următorii furnizori de date[2] (lista nu este completă):

Tabelul 8.1: Furnizori de date.

Nume furnizor	Prefix API	Descriere
ODBC Data Provider	Odbc	Surse de date cu interfaţă ODBC (baze de date vechi)
OleDb Data Provider	OleDb	Surse de date care expun o interfaţă OleDb, de exemplu Access şi Excel sau SQL Sever versiune mai veche de 7.0
Oracle Data Provider	Oracle	SGBD Oracle
SQL Data Provider	Sql	Pentru interacţiune cu Microsoft SQL Server 7.0, 2000, 2005 sau MSDE
Borland Data Provider	Bdp	Acces generic la SGBD-uri precum Interbase, SQL Server, IBM DB2, Oracle

Prefixele trecute în coloana "Prefix" sunt folosite pentru clasele de lucru specifice unui anumit furnizor ADO.NET: de exemplu, pentru o connexiune SQL se va folosi clasa SqlConnection.

8.3 Componentele unui furnizor de date

Fiecare furnizor de date ADO.NET constă în patru componente: *Connection*, *Command*, *DataReader*, *DataAdapter* . Arhitectura ADO.NET este

[1] Adresa: http://www.microsoft.com/sql/msde/downloads/download.asp

[2] Engl: Data Providers

prezentată în figura 8.1

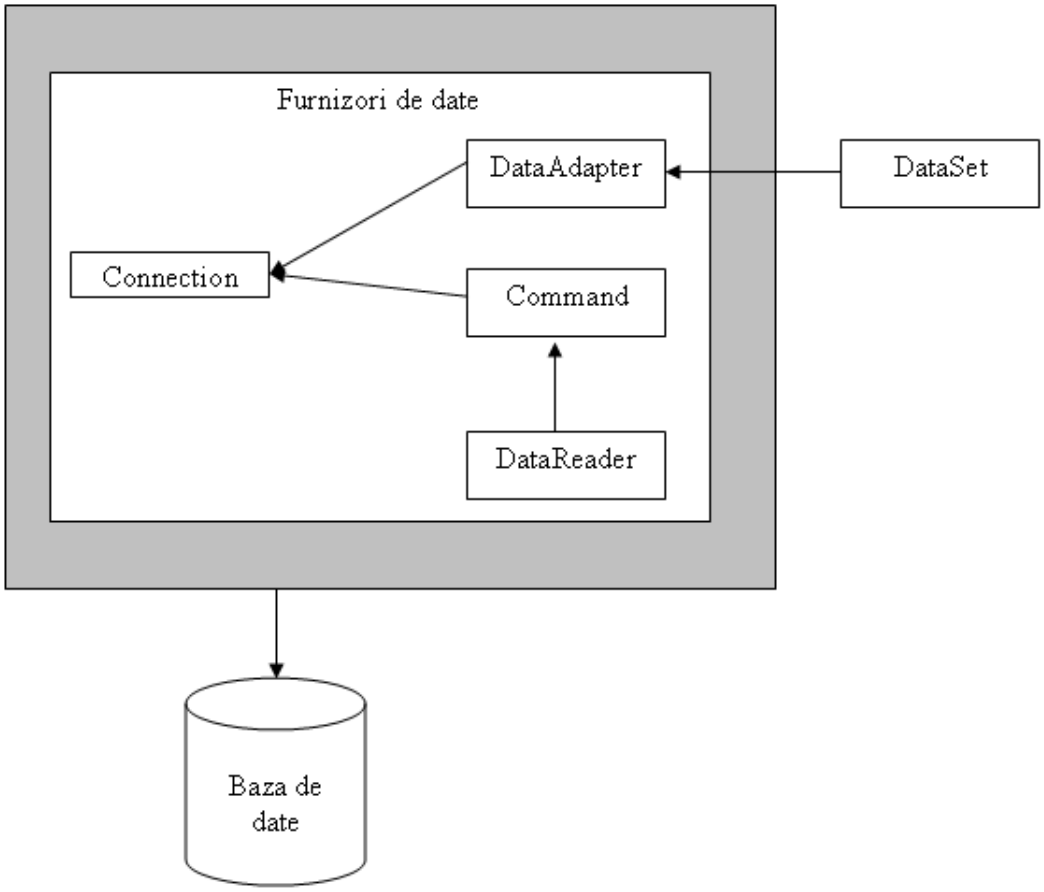

Figura 8.1: Principalele clase ADO.NET

Mai jos sunt date descrieri succinte ale claselor cel mai des utilizate.

8.3.1 Clasele *Connection*

Sunt folosite pentru a reprezenta o conexiune la surse de date specifice. Ele conţin date specifice conexiunii, cum ar fi locaţia sursei de date, numele şi parola contului de acces, etc. În plus au metode pentru deschiderea şi închiderea conexiunilor, pornirea unei tranzacţii sau setarea perioadei de time-out. Stau la baza oricărei încercări de conectare.

8.3.2 Clasele *Command*

Sunt folosite pentru a executa diferite comenzi pe baza de date (SELECT, INSERT, UPDATE, DELETE) şi pentru a furniza un obiect de tip *DataReader* sau *DataSet*. Pot fi folosite pentru apelarea de proceduri stocate aflate pe server. Ele permit scrierea de interogări SQL parametrizate sau specificarea parametrilor pentru procedurile stocate.

8.3.3 Clasele *DataReader*

Permit navigarea de tip forward–only, read–only în mod conectat la sursa de date. Se obţin pe baza unui obiect de tip *Command* prin apelul metodei *ExecuteReader()*. Accesul rezultat este extrem de rapid cu minim de resurse consumate.

8.3.4 Clasele *DataAdapter*

Ultima componentă principală a unui furnizor de date .NET este *DataAdapter*. Funcţionează ca o punte între sursa de date şi obiecte de tip *DataSet* deconectate, permiţând efectuarea operaţiilor pe baza de date. Conţin referinţe către obiecte de tip *Connection* şi deschid / închid singure conexiunea la baza de date. În plus, un *DataAdapter* conţine referinţe către patru comenzi pentru selectare, ştergere, modificare şi adăugare la baza de date.

8.3.5 Clasa *DataSet*

Această clasă nu este parte a unui furnizor de date .NET ci independentă de particularităţile de conectare şi lucru cu o bază de date anume. Prezintă marele avantaj că poate sa lucreze deconectat de la sursa de date, facilitând stocarea şi modificarea datelor local, apoi reflectarea acestor modificări în baza de date. Un obiect *DataSet* este de fapt un container de tabele şi relaţii între tabele. Foloseşte serviciile unui obiect de tip *DataAdapter* pentru aşi procura datele şi a trimite modificările înapoi către baza de date. Datele sunt stocate de un *DataSet* în format XML; acelaşi format este folosit pentru transportul datelor.

8.4 Obiecte *Connection*

Clasele de tip *Connection* pun la dispoziţie tot ceea ce e necesar pentru conectarea la baze de date. Este primul obiect cu care un programator ia contact atunci când încearcă să folosească un furnizor de date .NET. Înainte

ca o comandă să fie executată pe o bază de date trebuie stabilită o conexiune la ea.

Orice clasă de tip conexiune (din orice furnizor de date) implementează intefaţa *IDbConnection*. De exemplu, clasele *SqlConnection* (folosită pentru conectare la server Microsoft SQL Server 2000) sau *OleDbConnection* (folosită pentru conectare la fişiere .mdb din Access sau .xls) implementează *IDbConnection*.

Pentru deschiderea unei conexiuni se poate proceda ca mai jos:

```
using System.Data.SqlClient;//spatiul de nume SqlClient
...
SqlConnection cn = new SqlConnection("Data Source=serverBD;
Database=Northwind;User ID=sa;Password=parola");
cn.Open();
...
```

Mai sus s–a specificat numele calculatorului pe care se află instalat severul MSSQL de baze de date ("serverBD"), baza de date la care se face conectarea ("Northwind"), contul SQL cu care se face accesul ("sa") şi parola pentru acest cont ("parola").

Pentru conectarea la un fişier Access *Northwind.mdb* aflat în directorul c:\lucru se foloseşte un obiect de tipul OleDbConnection sub forma:

```
using System.Data.OleDb;//spatiul de nume OleDb
...
OleDbConnection cn = new OleDbConnection(
@"Provider=Microsoft.Jet.OLEDB.4.0;Data Source=
C:\Lucru\Northwind.mdb");
cn.Open();
...
```

S-a specificat furnizorul de date (Microsoft.Jet.OLEDB.4.0 pentru fişier Access) precum şi locul unde se află sursa de date (C:\Lucru\Northwind.mdb).

Vom enumera principalele proprietăţi, metode şi evenimente pentru un obiect de tip *Connection*.

8.4.1 Proprietăţi

1. *ConnectionString*: de tip *String*, cu accesori de *get* şi *set*; această proprietate defineşte un string valid care permite identificarea tipului şi locaţiei sursei de date la care se face conectarea şi eventual contul si

parola de acces. Acest string conţine lista de parametri necesari pentru conectare sub forma numeParametru=valoare, separaţi prin punct şi virgulă. Parametrii sunt:

- *provider*: se specifică furnizorul de date pentru conectarea la sursa de date. Acest furnizor trebuie precizat doar dacă se foloseşte OLE DB .NET Data Provider, însă nu se specifică pentru conectare la SQL Server.
- *Data Source*: se specifică numele serverului de baze de date sau numele fişierului de date.
- *Initial Catalog* (sinonim cu *Database*): specifică numele baze de date. Baza de date trebuie să se găsească pe serverul dat în *Data Source*.
- *User ID*: specifica un nume de utilizator care are acces de loginare la server.
- *Password*: specifică parola contului de mai sus.

2. *ConnectionTimeout*: de tip int, cu accesor de *get*, valoare implicită 15; specifică numărul de secunde pentru care un obiect de conexiune ar trebui să aştepte pentru realizarea conectării la server înainte de a se genera o excepţie. Se poate specifica o valoare diferită de 15 în *ConnectionString* folosind parametrul *Connect Timeout*:

```
SqlConnection cn = new SqlConnection("Data Source=serverBD;
Database=Northwind;User ID=sa;Password=parola;
Connect Timeout=30");
```

   Se poate specifica pentru *Connect Timeout* valoarea 0 cu semnificaţia "aşteaptă oricât", dar se sugerează să nu se procedeze în acest mod.

3. *Database*: atribut de tip string, read-only, returnează numele bazei de date la care s–a făcut conectarea. Folosită pentru a arăta unui utilizator care este baza de date pe care se face operarea.

4. *Provider*: atribut de tip string, read-only, returnează furnizorul OLE DB.

5. *ServerVersion*: atribut de tip string, read-only, returnează versiunea de server la care s–a făcut conectarea.

6. *State*: atribut de tip enumerare ConnectionState , read-only, returnează starea curentă a conexiunii. Valorile posibile sunt: *Broken*, *Closed*, *Connecting*, *Executing*, *Fetching*, *Open*.

8.4.2 Metode

1. *Open()*: deschide o conexiune la baza de date

2. *Close(), Dispose()*: închid conexiunea şi eliberează toate resursele alocate pentru ea

3. *BeginTransaction()*: pentru executarea unei tranzacţii pe baza de date; la sfârşit se apelează *Commit()* sau *Rollback()*.

4. *ChangeDatabase()*: se modifică baza de date la care se vor face conexiunile. Noua bază de date trebuie să existe pe acelaşi server ca precedenta.

5. *CreateCommand()*: creează un obiect de tip *Command* valid (care implementează interfaţa *IDbCommand*) asociat cu conexiunea curentă.

8.4.3 Evenimente

Un obiect de tip conexiune poate semnala două evenimente:

- evenimentul *StateChange*: apare atunci când se schimbă starea conexiunii. Event-handlerul este de tipul delegat *StateChangeEventHandler*, care spune care sunt stările între care s–a făcut tranziţia.

- evenimentul *InfoMessage*: apare atunci când furnizorul trimite un avertisment sau un mesaj informaţional către client.

8.4.4 Stocarea stringului de conexiune în fişier de configurare

În general este contraindicat ca stringul de conexiune să fie scris direct în cod; modificarea datelor de conectare (de exemplu parola pe cont sau sursa de date) ar însemna recompilarea codului.

.NET Framework permite menţinerea unor perechi de tipul chei-valoare, specifice aplicaţiei; fişierul este de tip XML. Pentru aplicaţiile Web fişierul se numeşte *web.config*, pentru aplicaţiile de tip consolă fişierul de configurare are extensia *config* şi numele aplicaţiei, iar pentru aplicaţiile Windows acest fişier nu există implicit, dar se adaugă prin click dreapta pe proiect în Solution Explorer->Add->Add New Item->Application Configuration File, implicit acesta având numele App.config. În fie care caz, structura fişierului XML de configurare este similară. Implicit, acesta arată astfel:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
</configuration>
```

În interiorrul elementului rădăcină *configuration* se va introduce elementul *appSettings*, care va conţine oricâte perechi cheie-valoare în interiorrul unui atribut numit *add*, precum mai jos:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="constring"
      value="Data Source=serverBD;database=Northwind;User ID=sa;
      pwd=parola"/>
  </appSettings>
</configuration>
```

Clasele neceasre pentru accesarea fişierului de configurare se găsesc în spaţiul de nume *System.Configuration*. Utilizarea stringului de conexiune definit anterior se face astfel:

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Configuration;
public class UsingConfigSettings
{
  public static void Main()
  {
    SqlConnection con = new SqlConnection(
    ConfigurationSettings.AppSettings["constring"]);
    //se lucreaza cu conexiunea
    con.Close();
  }
}
```

8.4.5 Gruparea conexiunilor

Gruparea conexiunilor[3] reprezintă reutilizarea resurselor de tip conexiune la o bază de date. Atunci când se creează o grupare de conexiuni se generează automat mai multe obiecte de tip conexiune, acest număr fiind

[3]Engl: connection pooling

egal cu minimul setat pentru gruparea respectivă. O nouă conexiune este creată dacă toate conexiunile sunt ocupate şi se cere o nouă conexiune. Dacă dimensiunea maxim setată a grupării este atinsă, atunci nu se va mai crea o conexiune nouă, ci se va pune într–o coadă de aşteptare. Dacă aşteparea durează mai mult decât este precizat în valoarea de Timeout se va arunca o excepţie. Pentru a returna o conexiune la grupare trebuie apelată metoda *Close()* sau *Dispose()* pentru acea conexiune.

Sursele de date .NET administrează automat acest aspect, degrevându-l pe programator de aceast aspect ce nu ţine de logica aplicaţiei. La dorinţă comportamentul implicit se poate modifica prin intemediul conţinutului stringului de conectare.

8.5 Obiecte *Command*

Un clasă de tip *Command* dată de un furnizor .NET trebuie să implementeze interfaţa *IDbCommand*, ca atare toate vor asigura un set de servicii bine specificat. Un asemenea obiect este folosit pentru a executa comenzi pe baza de date: SELECT, INSERT, DELETE, UPDATE sau apel de proceduri stocate (dacă SGBD-ul respectiv ştie acest lucru). Comanda se poate executa numai dacă s-a deschis o conexiune la baza de date.

Exemplu:

```
SqlConnection con = new SqlConnection(
  ConfigurationSettings.AppSettings["constring"]);
SqlCommand cmd = new SqlCommand("SELECT * FROM Customers", con);
```

Codul care utilizează o comandă pentru lucrul cu fisiere mdb sau xls ar fi foarte asemănător, cu diferenţa că în loc de *SqlCommand* se foloseşte *OleDbCommand*, din spaţiul de nume *System.Data.OleDb*. Nu trebuie modificat altceva, doarece locaţia sursei de date se specifică doar la conexiune.

Se observă că obiectul de conexiune este furnizat comenzii create. Enumerăm mai jos principalele proprietăţi şi metode ale unui obiect de tip comandă.

8.5.1 Proprietăţi

1. *CommandText*: de tip *String*, cu ambii accesori; conţine comanda SQL sau numele procedurii stocate care se execută pe sursa de date.

2. *CommandTimeout*: de tip int, cu ambii accesori; reprezintă numărul de secunde care trebuie să fie aşteptat pentru executarea interogării. Dacă se depăşeste acest timp, atunci se aruncă o excepţie.

3. *CommandType*: de tip *CommandType* (enumerare), cu ambii accesori; reprezintă tipul de comandă care se execută pe sursa de date. Valorile pot fi:

   - *CommandType.StoredProcedure* - interpretează comanda conţinută în proprietatea *CommandText* ca o un apel de procedură stocată definită în baza de date.
   - *CommandType.Text* - interpretează comanda ca fiind o comandă SQL clasică.
   - *CommandType.TableDirect* - momentan disponibil numai pentru furnizorul OLE DB (deci pentru obiect de tip OleDbCommand); dacă proprietatea *CommandType* are această valoare, atunci proprietatea *CommandText* este interpretată ca numele unui tabel pentru care se aduc toate liniile şi coloanele la momentul executării.

4. *Connection* - proprietate de tip *System. Data. [.NET Data Provider]. PrefixConnection*, cu ambii accesori; conţine obiectul de tip conexiune folosit pentru legarea la sursa de date; "Prefix" este prefixul asociat furnizorului respectiv (a se vedea tabelul 8.1).

5. *Parameters* - proprietate de tip *System.Data.[.NET Data Provider]. PrefixParameterCollection*, read-only; returnează o colecţie de parametri care s-au transmis comenzii; această listă a fost creată prin apelul metodei *CreateParameter()*."Prefix" reprezintă acelasi lucru ca mai sus.

6. *Transaction* - proprietate de tip *System.Data.[.NET Data Provider]. PrefixTransaction*, read-write; permite accesul la obiectul de tip tranzacţie care se cere a fi executat pe sursa de date.

8.5.2 Metode

1. *Constructori* - un obiect de tip comandă poate fi creat şi prin intermediul apelului de constructor; de exemplu un obiect *SqlCommand* se poate obţine astfel:

```
SqlCommand cmd;
cmd = new SqlCommand();
cmd = new SqlCommand(string CommandText);
cmd = new SqlCommand(string CommandText, SqlConnection con );
cmd = new SqlCommand(string CommandText, SqlConnection con,
     SqlTransaction trans);
```

2. *Cancel()* - încearcă să oprească o comandă dacă ea se află în execuţie. Dacă nu se află în execuţie, sau această încercare nu are nici un efect nu se intâmplă nimic.

3. *Dispose()* - distruge obiectul comandă.

4. *ExecuteNonQuery()* - execută o comandă care nu returnează un set de date din baza de date; dacă comanda a fost de tip INSERT, UPDATE, DELETE, se returnează numărul de înregistrări afectate. Dacă nu este definită conexiunea la baza de date sau aveasta nu este deschisă, se aruncă o excepţie de tip *InvalidOperationException*.

   Exemplu:

   ```
   SqlConnection con = new SqlConnection(
      ConfigurationSettings.AppSettings["constring"]);
   SqlCommand cmd = new SqlCommand();
   cmd.CommandText = "DELETE FROM Customers WHERE CustomerID = 'SEVEN'";
   cmd.Connection = con;
   con.Open();
   Console.WriteLine(cmd.ExecuteNonQuery().ToString());
   con.Close()
   ```

   În exemplul de mai sus se returnează numărul de înregistrări care au fost şterse.

5. *ExecuteReader()* - execută comanda conţinută în proprietatea *CommandText* şi se returnează un obiect de tip *IDataReader* (e.g. *SqlDataReader* sau *OleDbDataReader*).

   Exemplu: se obţine conţinutul tabelei Customers într–un obiect de tip *SqlDataReader* (se presupune că baza de date se stochează pe un server MSSQL):

   ```
   SqlConnection con = new SqlConnection(
     ConfigurationSettings.AppSettings["constring"]);
   SqlCommand cmd = new SqlCommand();
   cmd.CommandText = "SELECT * FROM Customers";
   cmd.Connection = con;
   con.Open();
   SqlDataReader reader = cmd.ExecuteReader();
   while(reader.Read())
   {
   ```

```
  Console.WriteLine("{0} - {1}",
  reader.GetString(0),
  reader.GetString(1));
}
reader.Close();
con.Close();
```

Metoda *ExecuteReader()* mai poate lua un argument opţional de tip enumerare *CommandBehavior* care descrie rezultatele şi efectul asupra bazei de date:

- *CommandBehavior.CloseConnection* - conexiunea este închisă atunci când obiectul de tip *IDataReader* este închis.
- *CommandBehavior.KeyInfo* - comanda returnează informaţie despre coloane şi cheia primară.
- *CommandBehavior.SchemaOnly* - comanda returnează doar informaţie despre coloane.
- *CommandBehavior.SequentialAccess* - dă posibilitatea unui *DataReader* să manipuleze înregistrări care conţin câmpuri cu valori binare de mare întindere. Acest mod permite încărcarea sub forma unui flux de date folosind *GetChars()* sau *GetBytes()*.
- *CommandBehavior.SingleResult* - se returnează un singur set de rezultate
- *CommandBehavior.SingleRow* - se returnează o singură linie. De exemplu, dacă în codul anterior înainte de *while* obţinerea obiectului *reader* s–ar face cu:

  ```
  SqlDataReader reader = cmd.ExecuteReader(
     CommandBehavior.SingleRow);
  ```

  atunci s–ar returna doar prima înregistrare din setul de date.

6. *ExecuteScalar()* - execută comanda conţinută în proprietatea *CommandText*; se returnează valoarea primei coloane de pe primul rând a setului de date rezultat; folosit pentru obţinerea unor rezultate de tip agregat ("SELECT COUNT(*) FROM CUSTOMERS", de exemplu).

7. *ExecuteXmlReader()* - returnează un obiect de tipul *XmlReader* obţinut din rezultatul interogării pe sursa de date.

   Exemplu:

```
SqlCommand custCMD=new SqlCommand("SELECT * FROM Customers
   FOR XML AUTO, ELEMENTS", con);
System.Xml.XmlReader myXR = custCMD.ExecuteXmlReader();
```

8.5.3 Utilizarea unei comenzi cu o procedură stocată

Pentru a se executa pe server o procedură stocată definită în baza respectivă, este necesar ca obiectul comandă să aibe proprietatea *CommandType* la valoarea *CommandType.StoredProcedure* iar proprietatea *CommandText* să conţină numele procedurii stocate:

```
SqlConnection con = new SqlConnection(
  ConfigurationSettings.AppSettings["constring"]);
SqlCommand cmd = new SqlCommand("Ten Most Expensive Products", con);
cmd.CommandType = CommandType.StoredProcedure;
con.Open();
SqlDataReader reader = cmd.ExecuteReader();
while(reader.Read())
{
 Console.WriteLine("{0} - {1}",
 reader.GetString(0), reader.GetDecimal(1));
}
reader.Close();
con.Close();
```

Observaţie: se remarcă în toate exemplele de mai sus faptul că fiecare conexiune se închide manual, printr-un apel de tipul `con.Close()`. Daca conexiunea a fost folosită pentru un obiect de tip *DataRead* atunci acesta din urmă trebuie să fie şi el închis, înaintea închiderii conexiunii. Dacă nu se face acest apel atunci conexiunea va fi ţinută ocupată şi nu va putea fi reutilizată.

8.5.4 Folosirea comenzilor parametrizate

Există posibilitatea de a rula cod SQL (interogări, proceduri stocate) pe server care să fie parametrizate. Orice furnizor de date .NET permite crearea obiectelor parametru care pot fi adăugate la o colecţie de parametri ai comenzii. Valoarea cestor parametrii se specifică fie prin numele lor (cazul *SqlParameter*), fie prin poziţia lor (cazul *OleDbParameter*).

Exemplu: vom aduce din tabela *Customers* toate înregistrările care au în câmpul *Country* valoarea “USA”.

```
SqlConnection con = new SqlConnection(
```

```
ConfigurationSettings.AppSettings["constring"]);
SqlCommand cmd = new
  SqlCommand("SELECT * FROM Customers WHERE Country=@country",con);
SqlParameter param = new SqlParameter("@country", SqlDbType.VarChar);
param.Value = "USA";
cmd.Parameters.Add( param );
con.Open();
SqlDataReader reader = cmd.ExecuteReader();
while(reader.Read())
{
  Console.WriteLine("{0} - {1}",
   reader.GetString(0), reader.GetString(1));
}
reader.Close();
con.Close();
```

Pentru parametrul creat s–a setat tipul lui (ca fiind tip SQL şir de caractere) şi valoarea. De reţinut faptul că numele parametrului se prefixează cu caracterul "@".

În cazul în care un parametru este de ieşire, acest lucru trebuie spus explicit folosind proprietatea *Direction* a parametrului respectiv:

```
SqlCommand cmd = new SqlCommand(
 "SELECT * FROM Customers WHERE Country = @country; " +
 "SELECT @count = COUNT(*) FROM Customers WHERE Country = @country",
 con);
SqlParameter param = new SqlParameter("@country", SqlDbType.VarChar);
param.Value = "USA";
cmd.Parameters.Add( param );
cmd.Parameters.Add(new SqlParameter("@count", SqlDbType.Int));
cmd.Parameters["@count"].Direction = ParameterDirection.Output;
con.Open();
SqlDataReader reader = cmd.ExecuteReader();
while(reader.Read())
{
  Console.WriteLine("{0} - {1}",
  reader.GetString(0),
  reader.GetString(1));
}
reader.Close();
Console.WriteLine("{0} - {1}", "Count",
  cmd.Parameters["@count"].Value.ToString());
```

```
con.Close();
```

Remarcăm următoarele:

- este posibil ca într–o comandă să se execute mai multe interogări
- pentru parametrul de ieşire numit "@count" trebuie făcută declarare de direcţie; implicit un parametru este de intrare
- parametrii de ieşire sunt accesibili doar după închiderea obiectului de tip *DataReader*

8.6 Obiecte *DataReader*

Un obiect de tip *DataReader* este folosit pentru a citi date dintr-o sursă de date. Caracteristicile unei asemenea clase sunt:

1. implementează întotdeauna interfaţa *IDataReader*
2. se lucrează conectat la sursa de date - pe toată perioada cât este accesat un *DataReader* necesită conexiune activă
3. este de tip read-only; dacă se doreşte modificarea datelor se poate folosi un *DataSet* + *DataAdapter*
4. este de tip forward-only - metoda de modificare a poziţiei curente este doar în direcţia înainte; orice reîntoarcere presupune reluarea înregistrărilor (dacă programatorul nu implementează el singur un mecanism de coadă)

Avantajele utilizării acestui tip de obiecte sunt: accesul conectat, performanţele bune, consumul mic de resurse şi tipizarea puternică.

8.6.1 Proprietăţi

1. *IsClosed* - proprietate read-only, returnează *true* daca obiectul este deschis, *false* altfel
2. *HasRows* - proprietate booleană read-only care spune dacă readerul conţine cel puţin o înregistrare
3. *Item* - indexator care dă acces la câmpurile unei înregistrări
4. *FieldCount* - dă numărul de câmpuri din înregistrarea curentă

8.6.2 Metode

1. *Close()* - închide obiectul de citire şi eliberează resursele client. Este obligatoriu apelul acestei metode ˆ inaintea închiderii conexiunii.

2. *GetBoolean(), GetByte(), GetChar(), GetDateTime(), GetDecimal(), GetDouble(), GetFloat(), GetInt16(), GetInt32(), GetInt64(), GetValue(), GetString()* returnează valorile câmpurilor din înergistrarea curentă. Preiau ca parametru indicele coloanei a cărei valoare se cere. *GetValue()* returnează un obiect de tip *Object*, pentru celelalte tipul returnat este descris de numele metodelor.

3. *GetBytes(), GetChars()* - returnează numărul de octeţi / caractere citiţi dintr–un câmp ce stochează o structură de dimensiuni mari; primeşte ca parametri indicele de coloană (int), poziţia din acea coloană de unde se va începe citirea, vectorul în care se face citirea, poziţia în buffer de la care se depun datele citite, numărul de octeţi/caractere ce urmează a fi citiţi.

4. *GetDataTypeName()* - returnează tipul coloanei specificat prin indice

5. *GetName()* - returnează numele coloanei

6. *IsDBNull()* - returnează *true* dacă în câmpul specificat prin index este o valoare de *NULL* (din baza de date)

7. *NextResult()* - determină trecerea la următorul rezultat, dacă aceasta există; în acest caz returnează *true*, altfel *false* (este posibil ca într–un *DataReader* să vină mai multe rezultate, provenind din interogări diferite)

8. *Read()* - determină trecerea la următoarea înregistrare, dacă aceasta existaă; în acest caz ea returnează *true*. Metoda trebuie chemată cel puţin o dată, deoarece iniţial poziţia curentă este înaintea primei înregistrări.

8.6.3 Crearea şi utilizarea unui *DataReader*

Nu este posibil să se creeze un obiect de tip *DataReader* cu ajutorul unui constructor, ci prin intermediul unui obiect de tip *Command*, folosind apelul *ExecuteReader()* (a se vedea secţiunea 8.3.2). Pentru comanda respectivă se specifică instrucţiunea care determină returnarea setului de date precum şi obiectul de conexiune. Această conexiune trebuie să fie deschisă înaintea apelului *ExecuteReader()*. Trecerea la următoarea înregistrare se

face folosind metoda *Read()*. Când se doreşte încetarea lucrului se închide reader–ul şi conexiunea. Omiterea închiderii obiectului de tip reader va duce la imposibilitatea reutilizării conexiunii iniţiale. După ce se închide acest *DataReader* este necesară şi în chiderea explicită a conexiunii (acest lucru nu mai e mandatoriu doar dacă la apelul metodei *ExecuteReader* s–a specificat *CommandBehavior.CloseConnection*). Dacă se încearcă refolosirea conexiunii fără ca readerul să fi fost închis se va arunca o excepţie *InvalidOperationException*.

Exemplu:

```
SqlConnection conn = new SqlConnection (
  ConfigurationSettings.AppSettings["constring"]);
SqlCommand selectCommand = new SqlCommand("select * from ORDERS", conn);
conn.Open ();
OleDbDataReader reader = selectCommand.ExecuteReader ( );
while ( reader.Read () )
{
  object id = reader["OrderID"];
  object date = reader["OrderDate"];
  object freight = reader["Freight"];
  Console.WriteLine ( "{0}\t{1}\t\t{2}", id, date, freight );
}
reader.Close ();
conn.Close ();
```

Este perfect posibil ca un obiect de tip *DataReader* să aducă datele prin apelul unei proceduri stocate (de fapt invocarea acestei proceduri este făcută de către obiectul de tip *Command*).

Următoarele observaţii trebuie luate în considerare atunci când se lucrează cu un obiect *DataReader*:

- Metoda *Read()* trebuie să fie întotdeauna apelată înaintea oricărui acces la date; poziţia curentă la deschidere este înaintea primei înregistrări.

- Întotdeauna apelaţi metoda *Close()* pe un *DataReader* şi pe conexiunea asociată cât mai repede posibil; în caz contrar conexiunea nu poate fi reutilizată

- Procesarea datelor citite trebuie să se facă după închiderea conexiunii; în felul acesta conexiunea se lasă liberă pentru a putea fi reutilizată.

8.6.4 Utilizarea de seturi de date multiple

Este posibil ca într–un *DataReader* să se aducă mai multe seturi de date. Acest lucru ar micşora numărul de apeluri pentru deschiderea unei conexiuni la stratul de date. Obiectul care permite acest lucru este chiar cel de tip *Command*:

```
string select = "select * from Categories; select * from customers";
SqlCommand command = new SqlCommand ( select, conn );
conn.Open ();
SqlDataReader reader = command.ExecuteReader ();
```

Trecerea de la un set de date la altul se face cu metoda *NextResult()* a obiectului de tip *Reader*:

```
do
{
  while ( reader.Read () )
  {
    Console.WriteLine ( "{0}\t\t{1}", reader[0], reader[1] );
  }
}while ( reader.NextResult () );
```

8.6.5 Accesarea datelor într–o manieră sigură din punct de vedere a tipului

Să considerăm următoarea secvenţă de cod:

```
while ( reader.Read () )
{
  object id = reader["OrderID"];
  object date = reader["OrderDate"];
  object freight = reader["Freight"];
  Console.WriteLine ( "{0}\t{1}\t\t{2}", id, date, freight );
}
```

După cum se observă, este posibil ca valorile câmpurilor dintr–o înregistrare să fie accesate prin intermediul numelui coloanei (sau a indicelui ei, pornind de la 0). Dezavantajul acestei metode este că tipul datelor returnate este pierdut (fiind returnate obiecte de tip *Object*), trebuind făcută un downcasting pentru a utiliza din plin facilităţile tipului respectiv. Pentru ca acest lucru să nu se întâmple se pot folosi metodele *GetXY* care returnează un tip specific de date:

```
while ( reader.Read () )
{
  int id = reader.GetInt32 ( 0 );
  DateTime date = reader.GetDateTime ( 3 );
  decimal freight = reader.GetDecimal ( 7 );
  Console.WriteLine ( "{0}\t{1}\t\t{2}", id, date, freight );
}
```

Avantajul secvenţei anterioare este că dacă se încearcă aducerea valorii unui câmp pentru care tipul nu este dat corect se aruncă o excepţie *InvalidCastException*; altfel spus, accesul la date se face sigur din punct de verere al tipului datelor.

Pentru a evita folosirea unor "constante magice" ca indici de coloană (precum mai sus: 0, 3, 7), se poate folosi următoarea strategie: indicii se obţin folosind apel de metodă *GetOrdinal* la care se specifică numele coloanei dorite:

```
private int OrderID;
private int OrderDate;
private int Freight;
...
OrderID = reader.GetOrdinal("OrderID");
OrderDate = reader.GetOrdinal("OrderDate");
Freight = reader.GetOrdinal("Freight");
...
reader.GetDecimal ( Freight );
...
```

Curs 9

ADO.NET (2)

9.1 Obiecte *DataAdapter*

La fel ca şi *Connection*, *Command*, *DataReader*, obiectele de tip *DataAdapter* fac parte din furnizorul de date .NET specific fiecărui tip de sursă de date. Scopul ei este să permită umplerea unui obiect *DataSet* cu date şi reflectarea schimbărilor efectuate asupra acestuia înapoi în baza de date (*DataSet* permite lucrul deconectat de la baza de date).

Orice clasă de tipul *DataAdapter* (de ex SqlDataAdapter şi OleDbDataAdapter) este derivată din clasa *DbDataAdapter* (clasă abstractă). Pentru orice obiect de acest tip trebuie specificată minim comanda de tip *SELECT* care care să populeze un obiect de tip *DataSet*; acest lucru este stabilit prin intermediul proprietăţii *SelectCommand* de tip *Command* (*SqlCommand*, *OleDbCommand*, ...). În cazul în care se doreşte şi modificarea informaţiilor din sursa de date (inserare, modificare, ştergere) trebuie specificate obiecte de tip comandă via proprietăţile: *InsertCommand*, *UpdateCommand*, *DeleteCommand*.

Exemplu: mai jos se preiau înregistrările din 2 tabele: *Authors* şi *TitleAuthor* şi se trec într–un obiect de tip *DataSet* pentru a fi procesate ulterior.

```
using System;
using System.Data;
using System.Data.SqlClient;

class DemoDataSource
{
  static void Main()
  {
    SqlConnection conn = new SqlConnection(
```

```
      ConfigurationSettings.AppSettings["constring"]);

    SqlDataAdapter daAuthors = new SqlDataAdapter("SELECT au_id,
      au_fname, au_lname FROM authors",conn);
    daAuthors.Fill(ds,"Author");

    SqlDataAdapter daTitleAuthor = new SqlDataAdapter("SELECT
      au_id, title_id FROM titleauthor", conn);
    daTitleAuthor.Fill(ds,"TitleAuthor");
  }
}
```

Prezentăm mai jos cele mai importante componente ale unei clase de tip *DataAdapter*.

9.1.1 Metode

1. *Constructori* de la cei impliciţi până la cei în care se specifică o comandă de tip *SELECT* şi conexiunea la sursa de date. Pentru un obiect de tip *SqlDataAdapter* se poate crea o instanţă în următoarele moduri:

   ```
   SqlDataAdapter da = new SqlDataAdapter();
   ```

   sau:

   ```
   SqlCommand cmd = new SqlCommand("SELECT * FROM Employees");
   SqlDataAdapter da = new SqlDataAdapter(cmd);
   ```

   sau:

   ```
   String strCmd = "SELECT * FROM Employees";
   String strConn = "...";
   SqlDataAdapter da = new SqlDataAdapter(strCmd, strConn);
   ```

2. *Fill()* – metodă polimorfică, permiţând umplerea unei tabele dintr–un obiect de tip *DataSet* cu date. Permite specificarea obiectului *DataSet* în care se depun datele, eventual a numelui tablei din acest *DataSet*, numărul de înregistrare cu care să se înceapă popularea (prima având indicele 0) şi numărul de înregistrări care urmează a fi aduse. Returnează de fiecare dată numărul de înregistrări care au fost aduse din bază. În clipa în care se apelează *Fill()* se procedează astfel:

(a) Se deschide conexiunea (dacă ea nu a fost explicit deschisă)

(b) Se aduc datele şi se populează un obiect de tip *DataTable* din *DataSet*

(c) Se închide conexiunea (dacă ea nu a fost explicit deschisă!)

De remarcat că un *DataAdapter* îşi poate deschide şi închide singur conexiunea, dar dacă aceasta a fost deschisă înaintea metodei *Fill()* atunci tot programatorul trebuie să o închidă.

3. *Update()* – metodă polimorfică, permiţând reflectarea modificărilor efectuate între–un *DataSet*. Pentru a funcţiona are nevoie de obiecte de tip comandă adecvate: proprietăţile *InsertCommand*, *DeleteCommand* şi *UpdateCommand* trebuie să indice către comenzi valide. Returnează de fiecare dată numărul de înregistrări afectate.

9.1.2 Proprietăţi

1. *DeleteCommand*, *InsertCommand*, *SelectCommand*, *UpdateCommand* – de tip *Command*, conţin comenzile ce se execută pentru selectarea sau modificarea datelor în sursa de date. Măcar proprietatea *SelectCommand* trebuie să indice către un obiect valid, pentru a se putea face popularea setului de date.

2. *MissingSchemaAction* – de tip enumerare *MissingSchemaAction*, determină ce se face atunci când datele care sunt aduse nu se potrivesc peste schema tablei în care sunt depuse. Poate avea următoarele valori:

   - *Add* - implicit, *DataAdapter* adaugă coloana la schema tablei
   - *AddWithKey* - ca mai sus, dar adaugă şi informaţii relativ la cine este cheia primară
   - *Ignore* - se ignoră lipsa coloanei respective, ceea ce duce la pierdere de date pe *DataSet*
   - *Error* - se generează o excepţie de tipul *InvalidOperationException*.

9.2 Clasa *DataSet*

Clasa *DataSet* nu mai face parte din biblioteca unui furnizor de date ADO.NET, fiind standard. Ea poate să conţină reprezentări tabelare ale datelor din bază precum şi diferite restricţii şi relaţii existente. Marele ei

avantaj este faptul că permite lucrul deconectat de la sursa de date, eliminând necesitatea unei conexiuni permanente precum la *DataReader*. În felul acesta, un server de aplicaţii sau un client oarecare poate apela la serverul de baze de date doar când preia datele sau când se doreşte salvarea lor. Funcţionează în strânsă legătură cu clasa *DataAdapter* care acţionează ca o punte între un *DataSet* şi sursa de date. Remarcabil este faptul că un *DataSet* poate face abstracţie de sursa de date, procesarea datelor desfăşurându–se independent de ea.

Figura 9.1 conţine o vedere parţială asupra clasei *DataSet*.

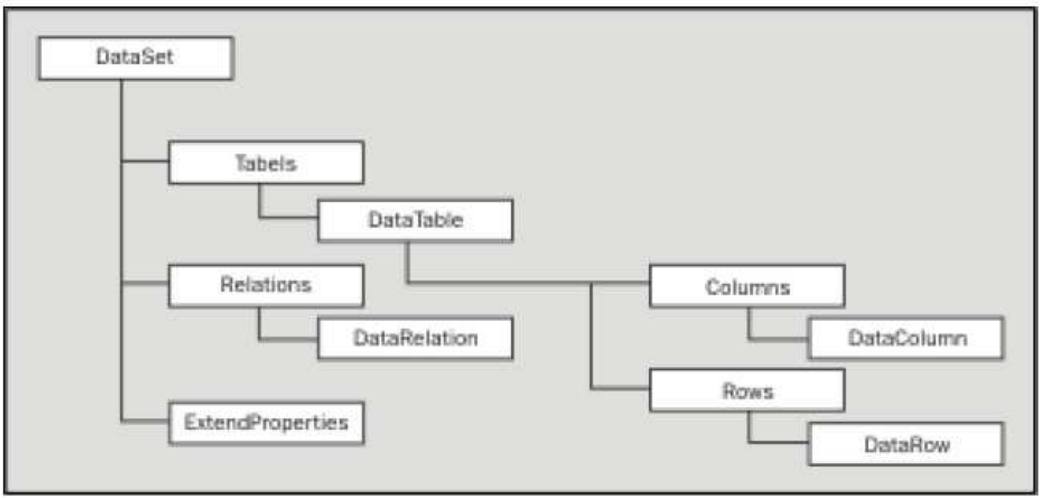

Figura 9.1: Structura unui *DataSet*

9.2.1 Conţinut

Prezentăm succint conţinutul unui *DataSet*:

1. Colecţia *Tables* conţine 0 sau mai multe obiecte *DataTable*. Fiecare *DataTable* este compusă dintr-o colecţie de linii şi coloane.

2. Colecţia *Relations* conţine 0 sau mai multe obiecte de tip *DataRelation*, folosite pentru marcarea legăturilor părinte–copil.

3. Colecţia *ExtendedProperties* conţine proprietăţi definite de utilizator.

9.2.2 Clasa *DataTable*

Datele sunt conţinute într-un *DataSet* sub forma unor tabele de tip *DataTable*. Aceste obiecte pot fi folosite atât independent, cât şi în interiorul unui

DataSet ca elemente ale colecţiei *Tables*. Un *DataTable* conţine o colecţie *Columns* de coloane, *Rows* de linii şi *Constraints* de constrângeri.

DataColumn

Un obiect *DataColumn* defineşte numele şi tipul unei coloane care face parte sau se adaugă unui obiect *DataTable*. Un obiect de acest tip se obţine prin apel de constructor sau pe baza metodei *DataTable.Columns.Add*.

Exemplu:

```
DataColumn myColumn = new DataColumn("title",
  Type.GetType("System.String"));
```

Definirea unei coloane ca fiind de tip autonumber (în vederea stabilirii ei ca şi cheie pe o tabelă) se face astfel:

```
DataColumn idColumn = new DataColumn("ID",
  Type.GetType("System.Int32"));
idColumn.AutoIncrement = true;
idColumn.AutoIncrementSeed = 1;
idColumn.ReadOnly = true;
```

DataRow

Un obiect de tip *DataRow* reprezintă o linie dintr-un obiect *DataTable*. Orice obiect *DataTable* conţine o proprietate *Rows* ce da acces la colecţia de obiecte *DataRow* conţinută. Pentru crearea unei linii se poate apela metoda *NewRow* pentru o tabelă a cărei schemă se cunoaşte. Mai jos este dată secvenţa de cod care creează o linie nouă pentru o tabelă şi o adaugă acestesia:

```
DataRow tempRow;
tempRow = myTable.NewRow();
tempRow["ID"] = 1;
tempRow["Name"] = "Book";
tempRow["Category"] = 1;
myTable.Rows.Add(tempRow);
```

Constrângeri

Constrângerile sunt folosite pentru a descrie anumite restricţii aplicate asupra valorilor din coloane. În ADO.NET există două tipuri de constrângeri: de unicitate şi de cheie străină. Toate obiectele de constrângere se află în colecţia *Constraints* a unei tabele.

- *UniqueConstraint* - precizează că într-o anumită coloană valorile sunt unice. Încercarea de a seta valori duplicate pe o coloana pentru care s-a precizat restricţia duce la aruncarea unei excepţii. Este necesară o asemenea coloană în clipa în care se foloseşte metoda *Find* pentru proprietatea *Rows*: în acest caz trebuie să se specifice o coloană pe care avem unicitate.

- *ForeignKeyConstraint* - specifică acţiunea care se va efectua atunci când se şterge sau modifică valoarea dintr–o anumită coloană. De exemplu se poate decide că dacă se sterge o înregistrare dintr-o tabelă atunci să se şteargă şi înregistrările copil. Valorile care se pot seta pentru o asemenea constrângere se specifică în proprietăţile *ForeignKeyConstraint.DeleteRule* şi *ForeignKeyConstraint.UpdateRule*:

  - *Rule.Cascade* - acţiunea implicită, şterge sau modifică înregistrările afectate
  - *Rule.SetNull* - se setează valoare de **null** pentru înregistrările afectate
  - *Rule.SetDefault* - se setează valoarea implicită definită în bază pentru câmpul respectiv
  - *Rule.None* - nu se execută nimic

  Exemplu:

```
ForeignKeyConstraint custOrderFK=new ForeignKeyConstraint
 ("CustOrderFK",custDS.Tables["CustTable"].Columns["CustomerID"],
  custDS.Tables["OrdersTable"].Columns["CustomerID"]);
custOrderFK.DeleteRule = Rule.None;
//Nu se poate sterge un client care are comenzi facute
custDS.Tables["OrdersTable"].Constraints.Add(custOrderFK);
```

  Mai sus s-a declarat o relaţie de tip cheie străină între două tabele ("CustTable" şi "OrdersTable", care fac parte dintr-un DataSet). Restricţia se adaugă la tabla copil.

Stabilirea cheii primare

O cheie primară se defineşte ca un vector de coloane care se atribuie proprietăţii *PrimaryKey* a unei tabele (obiect *DataTable*).

```
DataColumn[] pk = new DataColumn[1];
pk[0] = myTable.Columns["ID"];
myTable.PrimaryKey = pk;
```

Proprietatea *Rows* a clasei *DataTable* permite căutarea unei anumite linii din colecţia conţinută dacă se specifică un obiect sau un array de obiecte folosite pe post de cheie:

```
object key = 17;//cheia dupa care se face cautarea
DataRow line = myTable.Rows.Find(key);
if ( line != null )
  //proceseaza linia
```

9.2.3 Relaţii între tabele

Proprietatea *Relations* a unui obiect de tip *DataSet* conţine o colecţie de obiecte de tip *DataRelation* folosite pentru a figura relaţiile de tip părinte–copil între două tabele. Aceste relaţii se precizează în esenţă ca niste perechi de array-uri de coloane sau chiar coloane simple din cele două tabele care se relaţionează, de exemplu sub forma:

```
myDataSet.Relations.Add(DataColumn, DataColumn);
//sau
myDataSet.Relations.Add(DataColumn[], DataColumn[]);
```

concret:

```
myDataSet.Relations.Add(
  myDataSet.Tables["Customers"].Columns["CustomerID"],
  myDataSet.Tables["Orders"].Columns["CustomerID"]);
```

9.2.4 Popularea unui *DataSet*

Deşi un obiect *DataSet* se poate popula prin crearea dinamică a obiectelor *DataTable*, cazul cel mai des întâlnit este acela în care se populează prin intermediul unui obiect *DataAdapter*. O dată obţinut un asemenea obiect (care conţine cel puţin o comandă de tiop *SELECT*) se poate apela metoda *Fill()* care primeşte ca parametru *DataSet*-ul care se umple şi opţional numele tabelei care va conţine datele:

```
//defineste comanda de selectare din baza de date
String mySqlStmt ="SELECT * FROM Customers";
String myConString = ConfigurationSettings.AppSettings["constring"];
//Construieste obiectele de conexiune + comanda SELECT
SqlConnection myConnection = new SqlConnection(myConString);
SqlCommand myCommand = new SqlCommand(mySqlStmt, myConnection);
//Construieste obiectul DataAdapter
```

```
SqlDataAdapter myDataAdapter = new SqlDataAdapter();
//seteaza proprietatea SelectCommand pentru DataAdapter
myDataAdapter.SelectCommand = myCommand;
//construieste obiectul DataSet si il umple cu date
DataSet myDataSet = new DataSet();
myDataAdapter.Fill(myDataSet, "Customers");
```

Datele aduse mai sus sunt depuse într-un obiect de tip *DataTable* din interiorul lui *DataSet*, numit "Customers". Accesul la acest tabel se face prin construcţia

```
myDataSet.Tables["Customers"]
```

sau folosind indici întregi (prima tabelă are indicele 0). Acelaşi *DataSet* se poate popula în continuare cu alte tabele pe baza aceluiaşi sau a altor obiecte *DataAdapter*.

9.2.5 Propagarea modificărilor către baza de date

Pentru a propaga modificările efectuate asupra conţinutului tabelelor dintr-un *DataSet* către baza de date este nevoie să se definească adecvat obiecte comandă de tip *INSERT*, *UPDATE*, *DELETE*. Pentru cazuri simple se poate folosi clasa *SqlCommandBuilder* care va construi singură aceste comenzi.

Clasa *CommandBuilder*

Un obiect de tip *CommandBuilder* (ce provine din furnizorul de date) va analiza comanda *SELECT* care a adus datele în *DataSet* şi va construi cele 3 comenzi de update în funcţie de aceasta. E nevoie să se satisfacă 2 condiţii atunci când se uzează de un astfel de obiect:

1. Trebuie specificată o comandă de tip *SELECT* care să aducă datele dintr-o singură tabelă
2. Trebuie specificată cel puţin cheia primară sau o coloană cu constrângere de unicitate în comanda *SELECT*.

Pentru cea de a doua condiţie se poate proceda în felul următor: în comanda *SELECT* se specifică şi aducerea cheii, iar pentru obiectul *DataAdapter* care face aducerea din bază se setează proprietatea *MissingSchemaAction* pe valoarea *MissingSchemaAction.AddWithKey* (implicit este doar *Add*).

Fiecare linie modificată din colecţia *Rows* a unei tabele va avea modificată valoarea proprietăţii *RowState* astfel: *DataRowState.Added* pentru

o linie nouă adăugată, *DataRowState.Deleted* dacă e ştearsă şi *DataRowState.Modified* dacă a fost modificată. Apelul de update pe un dataReader va apela comanda necesară pentru fiecare linie care a fost modificată, în funcţie de starea ei.

Arătăm mai jos modul de utilizare a clasei *SqlCommandBuilder* pentru adăugarea, modificarea, ştergerea de înregistrări din baza de date.

```
SqlConnection conn = new SqlConnection(
  ConfigurationSettings.AppSettings["constring"]);
da = new SqlDataAdapter("SELECT id, name, address FROM
  customers",conn);
da.Fill(ds);

SqlCommandBuilder cb = new SqlCommandBuilder(da);
//determina liniile care au fost schimbate
DataSet dsChanges = ds.GetChanges();
if (dsChanges != null)
{
  // modifica baza de date
  da.Update(dsChanges);
  //accepta schimbarile din dataset
  ds.AcceptChanges();
}
```

În clipa în care se creează obiectul *SqlCommandBuilder* automat se vor completa proprietăţile *InsertCommand*, *DeleteCommand*, *UpdateCommand* ale dataAdapter-ului. Se determină apoi liniile care au fost modificate (prin interogarea stării lor) şi se obţine un nou *DataSet* care le va conţine doar pe acestea. Comanda de *Update* se dă doar pentru acest set de modificări, reducând astfel traficul spre serverul de baze de date.

Update folosind comenzi SQL

Atunci când interogările de aducere a datelor sunt mai complexe (de exemplu datele sunt aduse din mai multe table, printr-un join) se pot specifica propriile comenzi SQL prin intermediul proprietăţilor *InsertCommand*, *DeleteCommand UpdateCommand* ale obiectului *DataAdapter*. Pentru fiecare linie dintr-o tabelă care este modificată/adăugată/ştearsă se va apela comanda SQL corespunzătoare.Aceste comenzi pot fi fraze SQL parametrizate sau pot denumi proceduri stocate aflate în bază.

Să presupunem că s-a definit un *DataAdapter* legat la o bază de date. Instrucţiunea de selecţie este

```
SELECT CompanyName, Address, Country, CustomerID FROM Customers
```

unde *CustomerID* este cheia. Pentru inserarea unei noi înregistrări s–ar putea scrie codul de mai jos:

```
//da=obiect DataAdapter
da.InsertCommand.CommandText = @"INSERT INTO Customers (
  CompanyName, Address, Country) VALUES
  (@CompanyName, @Address, @Country);
  SELECT CompanyName, Address, Country, CustomerID FROM
  Customers WHERE (CustomerID = @@Identity)";
```

Update-ul efectiv se face prin prima instrucţiune de tip *Update*. Valorile pentru aceşti parametri se vor da la runtime, de exemplu prin alegerea lor dintr-un tabel. Valoarea pentru cheia CustomerID nu s-a specificat, deoarece (în acest caz) ea este de tip AutoNumber (SGBD-ul este cel care face managementul valorilor acestor câmpuri, nu programatorul). *@@Identity* reprezintă id-ul noii înergistrări adăugate în tabelă. Ultima instrucţiune va duce la reactualizarea obiectului *DataSet*, pentru ca acesta să conţină modificarea efectuată (de exemplu ar putea aduce valorile implicite puse pe anumite coloane).

Pentru modificarea conţinutului unei linii se poate declara instrucţiunea de *UPDATE* astfel:

```
da.UpdateCommand.CommandText = @"UPDATE Customers SET CompanyName
 = @CompanyName, Address = @Address, Country = @Country
 WHERE (CustomerID = @ID)";
```

Comanda de update este ca la cazul precedent.

9.3 Tranzacţii în ADO.NET

O tranzacţie este un set de operaţii care se efectuează fie în întregime, fie deloc. Să presupunem că se doreşte trecerea unei anumite sume de bani dintr-un cont în altul. Operaţia presupune 2 paşi:

1. scade suma din primul cont
2. adaugă suma la al doilea cont

Ar fi inadmisibil ca primul pas să reuşească iar al doilea să eşueze. Tranzacţiile satisfac nişte proprietăţi strânse sub numele ACID:

- **atomicitate** - toate operaţiile din tranzacţie ar trebui să aibă succes sau să eşueze împreună
- **consistenţă** - tranzacţia duce baza de date dintr-o stare stabilă în alta
- **izolare** - nici o tranzacţie nu ar trebui să afecteze o alta care rulează în acelaşi timp
- **durabilitate** - schimbările care apar în tipul tranzacţiei sunt permanent stocate pe un mediu.

Sunt trei comenzi care se folosesc în context de tranzacţii:

- *BEGIN* - înainte de executarea unei comenzi SQL sub o tranzacţie, aceasta trebuie să fie iniţializată
- *COMMIT* - se spune că o tranzacţie este terminată când toate schimbările cerute sunt trecute în baza de date
- *ROLLBACK* - dacă o parte a tranzacţiei eşuează, atunci toate operaţiile efectuate de la începutul tranzacţiei vor fi neglijate

Schema de lucru cu tranzacţiile sub ADO.NET este:

1. deschide conexiunea la baza de date
2. începe tranzacţia
3. execută comenzi pentru tranzacţie
4. dacă tranzacţia se poate efectua (nu sunt excepţii sau anumite condiţii sunt îndeplinite), efectuează *COMMIT*, altfel efectuează *ROLLBACK*
5. închide conexiunea la baza de date

Sub ADO.NET acest lucru s-ar face astfel:

```
SqlConnection myConnection = new SqlConnection(myConnString);
myConnection.Open();

SqlCommand myCommand1 = myConnection.CreateCommand();
SqlCommand myCommand2 = myConnection.CreateCommand();
SqlTransaction myTrans;

myTrans = myConnection.BeginTransaction();
//Trebuie asignate ambele obiecte: conexiune si tranzactie
```

```
//unui obiect de tip comanda care va participa la tranzactie
myCommand1.Transaction = myTrans;
myCommand2.Transaction = myTrans;

try
{
  myCommand1.CommandText = "Insert into Region (RegionID,
    RegionDescription) VALUES (100, 'Description')";
  myCommand1.ExecuteNonQuery();
  myCommand2.CommandText = "Insert into Region (RegionID,
    RegionDescription) VALUES (101, 'Description')";
  myCommand2.ExecuteNonQuery();
  myTrans.Commit();
  Console.WriteLine("Ambele inregistrari au fost scrise.");
}
catch(Exception e)
{
  myTrans.Rollback();
}
finally
{
  myConnection.Close();
}
```

Comanda de *ROLLBACK* se poate executa şi în alte situaţii, de exemplul comanda efectuată depăşteşte stocul disponibil.

Curs 10

Fire de execuţie

Firele de execuţie[1] sunt responsabile cu multitasking–ul în interiorul unei aplicaţii. Clasele şi interfeţele responsabile pentru crearea aplicaţiilor multithreading se găsesc în spaţiul de nume *System.Threading*. Vom discuta în cele ce urmează despre managementul thread–urilor şi despre sincronizare.

De cele mai multe ori crearea de thread–uri este necesară pentru a da utilizatorului impresia că programul execută mai multe acţiuni simultan, în cadrul unei aplicaţii. Pentru ca o interfaţă utilizator să poată fi folosită fără a se aştepta încheierea unei anumite secvenţe de instrucţiuni, e nevoie de mai multe fire de execuţie (unele pentru procesarea efectivă a informaţiei, altele care să răspundă acţiunilor utilizatorului). Pentru probleme de calcul numeric, exprimarea procesului de calcul se poate face foarte natural sub formă de fire de execţie.

10.1 Managementul thread–urilor

10.1.1 Pornirea thread–urilor

Cea mai simplă metodă de a crea un fir de execţie este de a crea o instanţă a clasei *Thread*, al cărei constructor preia un singur argument de tip delegat. În BCL este definit tipul delegat *ThreadStart*, care este folosit ca prototip pentru orice metodă care se vrea a fi lansată într–un fir de execuţie. Declaraţia de *ThreadStart* este:

```
public delegate void ThreadStart();
```

Crearea unui fir de execuţie se face pe baza unei metode care returnează *void* şi nu preia nici un parametru:

[1]Engl: threads; vom folosi alternativ termenii “thread” şi “fir de execuţie”

```
ThreadStart ts = new ThreadStart(MyFunc);
Thread myThread = new Thread( ts );
```

Un exemplu simplu este:

```
using System;
using System.Threading;

class SimpleThreadApp
{
    public static void WorkerThreadMethod()
    {
        Console.WriteLine("[WorkerThreadMethod] Worker " +
            "thread started");
    }

    public static void Main()
    {
        ThreadStart worker = new ThreadStart(WorkerThreadMethod);
        Console.WriteLine("[Main] Creating worker thread");
        Thread t = new Thread(worker);
        t.Start();
        Console.WriteLine( "[Main] Have requested the start of
         worker thread");
        Console.ReadLine();
    }
}
```

Până la linia `t.Start()` există un singur thread: cel dat de pornirea metodei `Main`. După `t.Start()` sunt 2 thread–uri: cel anterior şi `t`. Mesajul din Main va fi tipărit înaintea celui din thread–ul t; în funcţie de cum anume se planifică thread–urile pentru execuţie, mesajul de după t.Start() poate să apară înainte sau după mesajul tipărit de metoda `WorkerThreadMethod()`. De remarcat că simpla creere a firului de execuţie nu determină şi pornirea lui: acest lucru se întâmplă dupa apelul metodei `Start` definită în clasa `Thread`.

Exemplul următor porneşte două fire de execuţie. Funcţiile care conţin codul ce se va executa în câte un thread sunt `Increment()` şi `Decrement()`:

```
using System;
using System.Threading;
class Tester
```

```
{
  static void Main( )
  {
    Tester t = new Tester( );
    t.DoTest( );
  }
  public void DoTest( )
  {
    // creeaza un thread pentru Incrementer
    Thread t1 = new Thread( new ThreadStart(Incrementer) );
    // creeaza un thread pentru Decrementer
    Thread t2 = new Thread( new ThreadStart(Decrementer) );
    // porneste threadurile
    t1.Start( );
    t2.Start( );
  }

  public void Incrementer( )
  {
    for (int i = 1;i<=1000;i++)
    {
      Console.WriteLine( "Incrementer: {0}", i);
    }
  }

  public void Decrementer( )
  {
    for (int i = 1000;i>0;i--)
    {
      Console.WriteLine("Decrementer: {0}", i);
    }
  }
}
```

La ieşire se vor mixa mesajele tipărite de primul thread cu cele tipărite de cel de–al doilea thread:

```
...
Incrementer: 102
Incrementer: 103
Incrementer: 104
Incrementer: 105
```

```
Incrementer: 106
Decrementer: 1000
Decrementer: 999
Decrementer: 998
Decrementer: 997
...
```

Perioada de timp alocată fiecărui thread este determinatuă de către planificatorul de fire de execuţie[2] şi depinde de factori precum viteza procesorului, gradul lui de ocupare, etc.

O altă modalitate de obţinere a unei referinţe la un thread este prin apelul metodei statice `Thread.CurrentThread` pentru firul de execuţie curent. În exemplul de mai jos se obţine obiectul Thread asociat firului de execuţie curent, i se setează numele (proprietate de tip read/write) şi se afişează pe ecran:

```
Thread current = Thread.CurrentThread;
current.Name = "My name";
Console.WriteLine("nume={0}", current.Name );
```

10.1.2 Metoda *Join()*

Există situaţii în care, înaintea unei instrucţiuni trebuie să se asigure faptul că un alt fir de execuţie t s-a terminat; acest lucru se va face folosind apelul *t.Join()* înaintea instrucţiunii în cauză; în acest moment firul de execuţie care a apelat `t.Join()` intră în aşteptare.

Dacă de exemplu în metoda *Main* se lansează o colecţie de thread–uri (stocată în *myThreads*), atunci pentru a se continua execuţia numai după ce toate firele din colecţie s–au terminat, se procedează astfel:

```
foreach( Thread myThread in myThreads )
{
  myThread.Join();
}
Console.WriteLine(‘‘All my threads are done’’);
```

Mesajul final se va tipări doar când toate firele de execuţie s–au terminat.

[2]Engl: thread scheduler

10.1.3 Suspendarea firelor de execuţie

Se poate ca în anumite cazuri să se dorească suspendarea unui fir de execuţie pentru o scurtă perioadă de timp. Clasa *Thread* oferă o metodă statică supraîncărcată *Sleep()*, care poate prelua un parametru de tip *int* reprezentând milisecundele de "adormire", iar a doua variantă preia un argument de tip *TimeSpan*, care reprezintă cea mai mică unitate de timp care poate fi specificată, egală cu 100 nanosecunde. Pentru a cere firului de execuţie curent să se suspende pentru o secundă, se execută în cadrul acestuia:

```
Thread.Sleep(1000);
```

În acest fel se semnalează planificatorului de fire de execuţie că poate lansa un alt thread.

Dacă în exemplul de mai sus se adaugă un apel *Thread.Sleep(1)* după fiecare *WriteLine()*, atunci ieşirea se schimbă dramatic:

```
Iesire (extras)
Incrementer: 0
Incrementer: 1
Decrementer: 1000
Incrementer: 2
Decrementer: 999
Incrementer: 3
Decrementer: 998
Incrementer: 4
Decrementer: 997
Incrementer: 5
Decrementer: 996
Incrementer: 6
Decrementer: 995
```

10.1.4 Omorârea thread–urilor

De obicei, un fir de execuţie moare după ce se termină de executat. Se poate totuşi cere unui fir de execuţie să îşi înceteze execuţia folosind metoda *Abort()*. Acest lucru va duce la aruncarea unei excepţii în interiorul firului de execuţie căruia i se cere suspendarea: *ThreadAbortedException*, pe care firul respectiv o poate prinde şi procesa, permiţându–i eliberarea de resurse alocate. Ca atare, se recomandă ca o metodă care se va lansa ca fir de

execuţie să se compună dintr–un bloc *try* care conţine instrucţiunile utile, după care un bloc *catch* şi/sau *finally* care vor efectua eliberarea de resurse.

Exemplu:

```
using System;
using System.Threading;
class Tester
{
  static void Main( )
  {
    Tester t = new Tester( );
    t.DoTest( );
  }
  public void DoTest( )
  {
    // creeaza un vector de threaduri
    Thread[] myThreads =
    {
      new Thread( new ThreadStart(Decrementer) ),
      new Thread( new ThreadStart(Incrementer) ),
      new Thread( new ThreadStart(Incrementer) )
    };
    // porneste fiecare thread
    int ctr = 1;
    foreach (Thread myThread in myThreads)
    {
      myThread.IsBackground=true;
      myThread.Start( );
      myThread.Name = "Thread" + ctr.ToString( );
      ctr++;
      Console.WriteLine("Started thread {0}", myThread.Name);
      Thread.Sleep(50);
    }
    // dupa ce firele se pornesc,
    // comanda oprirea threadului 1
    myThreads[1].Abort( );
    // asteapta ca fiecare thread sa se termine
    foreach (Thread myThread in myThreads)
    {
      myThread.Join( );
    }
```

```
    Console.WriteLine("All my threads are done.");
  }
  // numara descrescator de la 1000
  public void Decrementer( )
  {
    try
    {
      for (int i = 1000;i>0;i--)
      {
        Console.WriteLine(``Thread {0}. Decrementer: {1}'',
        Thread.CurrentThread.Name, i);
        Thread.Sleep(1);
      }
    }
    catch (ThreadAbortedException)
    {
      Console.WriteLine(
      ``Thread {0} interrupted! Cleaning up...'',
      Thread.CurrentThread.Name);
    }
    finally
    {
      Console.WriteLine(``Thread {0} Exiting. '',
      Thread.CurrentThread.Name);
    }
  }
  // numara cresacator pana la 1000
  public void Incrementer( )
  {
    try
    {
      for (int i =1;i<=1000;i++)
      {
        Console.WriteLine(``Thread {0}. Incrementer: {1}'',
        Thread.CurrentThread.Name, i);
        Thread.Sleep(1);
      }
    }
    catch (ThreadAbortedException)
    {
      Console.WriteLine( ``Thread {0} interrupted! Cleaning up...'',
```

```
      Thread.CurrentThread.Name);
    }
    finally
    {
      Console.WriteLine( ‘‘Thread {0} Exiting. ’’,
      Thread.CurrentThread.Name);
    }
  }
}
```

Ieşire:

```
Started thread Thread1
Thread Thread1. Decrementer: 1000
Thread Thread1. Decrementer: 999
Thread Thread1. Decrementer: 998
Started thread Thread2
Thread Thread1. Decrementer: 997
Thread Thread2. Incrementer: 0
Thread Thread1. Decrementer: 996
Thread Thread2. Incrementer: 1
Thread Thread1. Decrementer: 995
Thread Thread2. Incrementer: 2
Thread Thread1. Decrementer: 994
Thread Thread2. Incrementer: 3
Started thread Thread3
Thread Thread1. Decrementer: 993
Thread Thread2. Incrementer: 4
Thread Thread2. Incrementer: 5
Thread Thread1. Decrementer: 992
Thread Thread2. Incrementer: 6
Thread Thread1. Decrementer: 991
Thread Thread3. Incrementer: 0
Thread Thread2. Incrementer: 7
Thread Thread1. Decrementer: 990
Thread Thread3. Incrementer: 1
Thread Thread2 interrupted! Cleaning up...
Thread Thread2 Exiting.
Thread Thread1. Decrementer: 989
Thread Thread3. Incrementer: 2
Thread Thread1. Decrementer: 988
Thread Thread3. Incrementer: 3
```

```
Thread Thread1. Decrementer: 987
Thread Thread3. Incrementer: 4
Thread Thread1. Decrementer: 986
Thread Thread3. Incrementer: 5
// ...
Thread Thread1. Decrementer: 1
Thread Thread3. Incrementer: 997
```

10.1.5 Sugerarea priorităţilor firelor de execuţie

Un fir de execuţie se lansează implicit cu prioritatea *ThreadPriorityLevel.Normal*. Dar scheduler–ul poate fi influenţat în activitatea sa prin setarea de diferite nivele de prioritate pentru fire; aceste nivele fac parte din enumerarea *ThreadPriorityLevel*: *ThreadPriorityLevel.TimeCritical, ThreadPriorityLevel.Highest, ThreadPriorityLevel.AboveNormal, ThreadPriorityLevel.Normal, ThreadPriorityLevel.BelowNormal, ThreadPriorityLevel.Lowest, ThreadPriorityLevel.Idle*. Prioritatea este descrescătoare în lista prezentată. Prioritatea este descrescătoare în lista prezentată. Pe baza priorităţii procesului care conţine firele de execuţie şi a priorităţii firelor, se calculează un nivel de propritate (de ex. pe maşini Intel valori între 0 şi 31) care determină prioritatea în ansamblul sistemului de operare a firului respectiv.

Setarea unei anumite priorităţi se face folosind proprietatea *Priority*:

```
myThread.Priority = ThreadPriorityLevel.Highest;
```

10.1.6 Fire în fundal şi fire în prim-plan

Relativ la proprietatea booleană `IsBackground`, trebuie făcută precizarea că un fir de execuţie poate să se execute în fundal (background) sau în prim plan (foreground). Diferenţa dintre cele două posibilităţi o constituie faptul că dacă un proces are măcar un fir de execuţie în foreground, CLR va menţine aplicaţia în execuţie. O dată ce toate firele de execuţie de tip foreground se termină, CLR va executa Abort() pentru fiecare fir de execuţie de tip background (dacă mai există aşa ceva) şi termină procesul.

Exemplu:

```
using System;
using System.Threading;

class Test
{
    static void Main()
```

```
    {
        BackgroundTest shortTest = new BackgroundTest(10);
        Thread foregroundThread =
            new Thread(new ThreadStart(shortTest.RunLoop));
        foregroundThread.Name = "ForegroundThread";

        BackgroundTest longTest = new BackgroundTest(50);
        Thread backgroundThread =
            new Thread(new ThreadStart(longTest.RunLoop));
        backgroundThread.Name = "BackgroundThread";
        backgroundThread.IsBackground = true;

        foregroundThread.Start();
        backgroundThread.Start();
    }
}

class BackgroundTest
{
    int maxIterations;

    public BackgroundTest(int maxIterations)
    {
        this.maxIterations = maxIterations;
    }

    public void RunLoop()
    {
        String threadName = Thread.CurrentThread.Name;

        for(int i = 0; i < maxIterations; i++)
        {
            Console.WriteLine("{0} count: {1}",
                threadName, i.ToString());
            Thread.Sleep(250);
        }
        Console.WriteLine("{0} finished counting.", threadName);
    }
}
```

Firul din foreground va menţine procesul în execuţie până când se termină

ciclul său `while`. Când acesta se termină, procesul este oprit, chiar dacă ciclul `while` din firul de execuţie din background nu şi-a terminat execuţia.

10.2 Sincronizarea

Sincronizarea se ocupă cu controlarea accesului la resurse partajate de mai multe fire de execuţie. De exemplu, se poate cere ca utilizarea unei resurse anume să se facă la un moment dat de către un singur fir de execuţie. Vom discuta aici trei mecanisme de sincronizare: clasa *Interlock*, instrucţiunea C# *lock* şi clasa *Monitor*. Exemplele se vor baza pe acces la o resursă partajată, ca mai jos:

```
public void Incrementer( )
{
  try
  {
    while (counter < 1000)
    {
      int temp = counter;
      temp++; // increment
      // simuleza o sarcina oarecare in acest thread
      Thread.Sleep(1);
      // atribuie valoarea incrementata
      // variabilei counter
      // si afiseaza rezultatul
      counter = temp;
      Console.WriteLine( ‘‘Thread {0}. Incrementer: {1}’’,
      Thread.CurrentThread.Name, counter);
    }
  }
  catch (ThreadAbortedException)
  {
    Console.WriteLine( ‘‘Thread {0} interrupted! Cleaning up...’’,
    Thread.CurrentThread.Name);
  }
  finally
  {
    Console.WriteLine( ‘‘Thread {0} Exiting. ’’,
    Thread.CurrentThread.Name);
  }
}
```

Câmpul *counter* se iniţializează cu 0. Să presupunem că pornim două fire de execuţie pe baza metodei *Incrementer()* de mai sus. Este posibil să se întâmple următoarele: primul thread va citi valoarea lui *counter* (0) şi o va atribui unei variabile temporare, pe care o va incrementa apoi. Al doilea fir se activează şi el, va citi valoarea (nemodificată) lui *counter* şi va atribui această valoare unei variabile temporare. Primul fir de execuţie îşi termină munca, apoi asignează valoarea variabilei temporare (1) lui *counter* şi o afişează. Al doilea thread face exact acelaşi lucru. Se tipăreşte astfel 1, 1. La următoarele iteraţii se va afişa 2, 2, 3, 3, etc, în locul lui 1, 2, 3, 4.

Exemplu:

```
using System;
using System.Threading;
class Tester
{
  private int counter = 0;
  static void Main( )
  {
    Tester t = new Tester( );
    t.DoTest( );
  }
  public void DoTest( )
  {
    Thread t1 = new Thread( new ThreadStart(this.Incrementer) );
    t1.IsBackground=true;
    t1.Name = ‘‘ThreadOne’’;
    t1.Start( );
    Console.WriteLine(‘‘Started thread {0}’’, t1.Name);
    Thread t2 = new Thread( new ThreadStart(this.Incrementer) );
    t2.IsBackground=true;
    t2.Name = ‘‘ThreadTwo’’;
    t2.Start( );
    Console.WriteLine(‘‘Started thread {0}’’, t2.Name);
    t1.Join( );
    t2.Join( );
  }
  // numara crescator pana la 1000
  public void Incrementer( )
  {
    //la fel ca la inceputul sectiunii
  }
```

```
}
```

Ieşire:

```
Started thread ThreadOne
Started thread ThreadTwo
Thread ThreadOne. Incrementer: 1
Thread ThreadOne. Incrementer: 2
Thread ThreadOne. Incrementer: 3
Thread ThreadTwo. Incrementer: 3
Thread ThreadTwo. Incrementer: 4
Thread ThreadOne. Incrementer: 4
Thread ThreadTwo. Incrementer: 5
Thread ThreadOne. Incrementer: 5
Thread ThreadTwo. Incrementer: 6
Thread ThreadOne. Incrementer: 6
```

Trebuie deci să se realizeze o excludere reciprocă a thread–urilor pentru accesul la *counter*.

10.2.1 Clasa *Interlocked*

Incrementarea şi decrementarea unei valori este o situaţie atât de des întâlnită, încât C# pune la dispoziţie o clasă specială *Interlocked* pentru o rezolvare rapidă. Clasa include două metode statice, *Increment()* şi *Decrement()*, care incrementează sau decrementează o valoare, însă sub un control sincronizat.

Putem modifica metoda *Incrementer()* de mai sus astfel:

```
public void Incrementer( )
{
  try
  {
    while (counter < 1000)
    {
      Interlocked.Increment(ref counter);
      // simuleaza o sarcina in aceasta metoda
      Thread.Sleep(1);
      // asigura valoarea decrementata
      // si afiseaza rezultatul
      Console.WriteLine(
      "Thread {0}. Incrementer: {1}",
```

```
      Thread.CurrentThread.Name,
      counter);
    }
  }
  //blocurile catch si finally raman neschimbate
}
```

Ieşirea este cea dorită:

```
Started thread ThreadOne
Started thread ThreadTwo
Thread ThreadOne. Incrementer: 1
Thread ThreadTwo. Incrementer: 2
Thread ThreadOne. Incrementer: 3
Thread ThreadTwo. Incrementer: 4
Thread ThreadOne. Incrementer: 5
Thread ThreadTwo. Incrementer: 6
Thread ThreadOne. Incrementer: 7
Thread ThreadTwo. Incrementer: 8
```

10.2.2 Instrucţiunea *lock*

Există situaţii când vrem să blocăm alte variabile decât cele de tip int. Un lock marchează o secţiune critică a codului, producând astfel sincronizare pentru un obiect. La utilizare, se specifică un obiect pentru care se stabileşte un lock, după care o instruţiune sau un grup de instrucţiuni. Lock–ul este înlăturat la sfârşitul instrucţiunii/blocului de instruţiuni. Sintaxa este:

```
lock(expresie)
{
  instructiuni
}
```

Exemplu: metoda *Incrementer()* se va modifica după cum urmează:

```
public void Incrementer( )
{
  try
  {
    while (counter < 1000)
    {
      lock (this)
      {
```

```
        int temp = counter;
        temp++;
        Thread.Sleep(1);
        counter = temp;
      }
      // atribuie valoarea decrementata
      // si afiseaza rezultatul
      Console.WriteLine( "Thread {0}. Incrementer: {1}",
      Thread.CurrentThread.Name, counter);
    }
  }
  //blocurile catch si finally raman neschimbate
}
```

Rezultatele sunt afişate exact ca la secţiunea 10.2.1.

10.2.3 Clasa *Monitor*

Clasa *Monitor* conţine metode pentru a controla sincronizarea firelor de execuţie, permiţând declararea unei zone critice în care la un moment dat doar un thread trebuie să opereze.

Atunci când se doreşte să se înceapă sincronizarea, se va apela metoda *Enter*, dând obiectul pentru care se va face blocarea:

```
Monitor.Enter( obiect );
```

Dacă monitorul este nedisponibil, atunci înseamnă că un alt thread este într–o regiune critică a obiectului respectiv. Se mai poate folosi de asemenea metoda *Wait()*, care eliberează monitorul, dar blochează threadul, informând CLR că atunci când monitorul devine din nou liber, thread–ul curent ar vrea să îşi continue execuţia (este adăugat într–o coadă de aşteptare formată din fire de execuţie blocate pe obiect). Terminarea zonei critice se face folosind metoda *Exit()* a clasei *Monitor*. Metoda *Pulse()* semnalează că a avut loc o schimbare de stare, în urma căreia este posibil ca un alt fir de execuţie care aşteaptă va putea să fie continuat (ordinea de selectare a firelor ce vor fi executate fiind ordinea introducerii în coada de aşteptare). Înrudită este metoda *PulseAll* care anunţă toate obiectele blocate pe un anumit obiect de schimbarea de stare.

Să presupunem că avem o clasă *MessageBoard* unde fire individuale pot citi şi scrie mesaje. Vom sincroniza accesul la această clasă astfel încât doar un thread să poată acţiona la un moment dat. Clasa *MessageBoard* va avea o metodă *Reader()* şi una *Writer()*.

Metoda *Reader()* determină dacă string–ul *message* conţine vreun mesaj valabil, iar metoda *Writer()* va scrie în acest string. Dacă nu sunt mesaje în timpul citirii, thread–ul *Reader()* va intra în stare de aşteptare folosind *Wait()* până când metoda *Writer()* scrie un mesaj şi transmite un mesaj via *Pulse()* pentru a trezi alte thread–uri.

```
using System;
using System.Threading;
class MessageBoard
{
  private String messages = ‘‘no messages’’ ;
  public void Reader()
  {
    try
    {
      Monitor.Enter(this);
      //daca nu e nici un mesaj atunci asteapta
      if (messages == ‘‘no messages’’)
      {
        Console.WriteLine(‘‘{0} {1}’’,
        Thread.CurrentThread.Name, messages);
        Console.WriteLine(‘‘{0} waiting...’’,
        Thread.CurrentThread.Name);
        Monitor.Wait(this);
      }
      //inseamna ca mesajul s-a schimbat
      Console.WriteLine(‘‘{0} {1}’’,
      Thread.CurrentThread.Name, messages);
    }
    finally
    {
      Monitor.Exit(this);
    }
  }
  public void Writer()
  {
    try
    {
      Monitor.Enter(this);
      messages = ‘‘Greetings Caroline and Marianne!’’;
      Console.WriteLine(‘‘{0} Done writing message...’’,
```

```
      Thread.CurrentThread.Name);
      //semnaleaza threadului de asteptare ca s-a schimbat mesajul
      Monitor.Pulse(this);
    }
    finally
    {
      Monitor.Exit(this);
    }
  }
  public static void Main()
  {
    MessageBoard myMessageBoard = new MessageBoard();
    Thread reader = new Thread(new
    ThreadStart(myMessageBoard.Reader));
    reader.Name = ``ReaderThread:'';
    Thread writer = new Thread( new
    ThreadStart(myMessageBoard.Writer));
    writer.Name = ``WriterThread:'';
    reader.Start();
    writer.Start();
  }
}
```

Ieşirea este:

```
ReadrThread: no messages
ReaderThread: waiting...
WriterThread: Done writing message...
ReaderThread: Greetings Caroline and Marianne!
```

După cum se vede mai sus, thread–ul *reader* este pornit primul, el blochează clasa obeictul de tip *MessageBoard*, ceea ce înseamnă că are acces exclusiv la variabila *message*. Dar deoarece *message* nu conţine nici un mesaj, thread-ul *reader* eliberează obiectul pe care l–a blocat mai înainte prin apelul *Wait()*. Thread–ul *writer* va putea acum să blocheze obiectul pentru a scrie mesajul. Apoi el cheamă metoda *Pulse()* pentru a semnala firului *reader* că a apărut o schimbare de stare a obiectului indicat wde *this*.

Curs 11

ASP.NET

ASP.NET (actualmente versiunea 1.1) permite crearea de aplicaţii Web folosind platforma .NET Framework; este una din cele 3 tipuri de aplicaţii creabile sub această platformă, alături de Windows Forms şi Web Services. Este destina a înlocui vechiul ASP (Active Server Pages), fiind obiectual şi integrat cu toate elementele constitutive ale lui .NET Framework: CLR, BCL, CTS, FCL, etc. Se poate rula pe servere de aplicaţii IIS (versiuni 5, 6), pe serverul open-source Cassini şi de asemenea poate fi integrat cu Apache (folosind Cassini). Rezultatele cele mai bune se obţin totuşi pe IIS 6.0 (integrat în Windows 2003 Server).

Enunţăm mai jos o listă neexhaustivă a trăsăturilor ASP.NET:

- unul din marile avantaje ale lui ASP.NET faţă de ASP este faptul că codul este păstrat compilat (sub formă de cod .NET PE, executat de către CLR) în loc ca pagina să fie interpretată la fiecare rulare;

- modul de realizare este orientat pe obiecte, o aplicaţie fiind compusă din două părţi: pagina afişată în browser şi pagina de "code-behind", care este o clasă; acest code-behind are posibilitatea de accesare a bazelor de date sau chiar a codului unmanaged (COM-uri şi DLL-uri anterior existente); în plus, se realizează separarea părţii de interfaţă utilizator de ceea ce este executat pe server;

- pune la dispoziţie un mod de tratare a erorilor unificat (tratarea de excepţii) şi foloseşte controale Web predefinite care se pot executa pe server sau pe client;

- are un mecanism încorporat care permite păstrarea stării paginii (chiar dacă prototlul de comunicare HTTP este fără stare, stateless);

- ASP.NET permite includerea de mai multe funţionalităţi pe controalele ASP.NET iar managementul stării lor este mult îmbunătăţit
- permite programarea bazată pe evenimente, cu diferenţe minime faţă de programarea aplicaţiilor windows;
- este senzitiv la browserul pentru care se face trimiterea codului HTML, alegând automat codul optimizat pentru o colecţie de browsere
- erorile grave apărute în aplicaţie ASP.NET nu vor duce la oprirea serverului Web (dar pot duce la imposibilitatea de a mai executa pagini ASP.NET până la repornirea serverului)
- permite scrierea codului care se va executa pe server în orice limbaj .NET: C#, VB.NET, etc

11.1 Anatomia unei pagini ASP.NET

O pagină ASP.NET se poate crea înntr-o multitudine de feluri, de exemplu prin Notepad, Web Matrix sau VS.NET (varianta recomandată, de altfel). Se poate crea o pagină simplă pe baza următorului conţinut:

```
<html>
  <head>
  </head>
  <body>
    Hello World
  </body>
</html>
```

salvat sub numele de HelloWorld.aspx în directorul rădăcină al site-ului web implicit (de obicei c:\Inetpub\wwwroot, creat automat la instalarea IIS-ului). Pagina se încarcă în browser folosind adresa http://localhost/HelloWorld.aspx.

11.1.1 Adăugarea unui control Web

Într-o pagină Web se poate adăuga un control pe pagină care să fie generat de către serverul IIS în mod adecvat şi care poate fi accesat programatic prin cod. Astfel, în pagina de mai sus se adaugă astfel încât să devină:

```
<html>
  <head>
  </head>
```

```
  <body>
    <form runat=server>
      <asp:Label text="Hellow World" runat=server />
    </form>
  </body>
</html>
```

Dacă se mai încarcă o dată pagina, se va genera următorul cod HTML pentru browser (vizibil prin opţiunea View Source):

```
<html>
  <head>
  </head>
  <body>
    <form name="_ct10" method="post" action="HelloWorld2.aspx" id="_ct10">
      <input type="hidden" name="_VIEWSTATE" value="dDw5MjMzODA0MjI7Oz4="/>
      <span>Hello World</span>
    </form>
  </body>
</html>
```

În sursa s-a încorporat un control Web `<asp:Label>` care prezintă proprietatea *Text* cu textul care va fi afşat în pagină. Atributul *runat* are ca valoare cuvântul *server* (singura opţiune posibilă) care spune că respectivul control se va procesa pe server şi nu pe client. După cum se vede în sursa HTML generată, acest control va avea ca rezultat un tag de tip *span*. De remarcat că controlul Web a fos scris în interiorul unui formular; acest lucru este mandatoriu pentru toate controalele Web sau HTML.

11.1.2 Adăugarea scriptului *inline*

Vom adăuga la pagina anterioară un buton a cărui apăsare va duce la schimbarea textului etichetei anterioare; această schimbare va fi făcută pe server prin apelul unei metode scrise în C#:

```
<html>
  <head>
  </head>
  <body>
    <form method="post" runat="server">
      <asp:Label id=lblHelloWorld text="Hello World" runat="Server"/>
      <br>
```

```
      <asp:Button onclick="ClickedIt" text="Submit" runat=Server />
    </form>
  </body>
  <script Language=C# runat=server>
    void ClickedIt(object sender, System.EventArgs e)
    {
      lblHelloWorld.Text = "Text schimbat";
    }
  </script>
</html>
```

La apăsarea butonului se va face submiterea formularului către server, spre aceeaşi pagină HelloWorld.aspx (lucru care se poate verifica din inspectarea codului HTML generat). Pentru a se putea accesa controlul de tip Label pe partea de server, trebuie ca acestuia să i se asigneze un ID (lblHelloWorld în cazul nostru). Pentru controlul Button (reprezentat de tag-ul `<asp:Button\>`) s-au asignat textul şi metoda care se va apela pe server (ClickedIt). Codul pentru ClickedIt este scris inline în aceeaşi pagină ASP.NET (pentru simplitate, dar vom vedea că această metodă are o variantă mai bună). Remarcăm că dacă s-a apăsat pe buton apoi se face refresh pe browser se va retransmite "Text schimbat", deoarece serverul menţine starea controalelor.

Cea mai importantă este observaţia că la codul HTML generat pe client s-a adăugat un element input de tip hidden cu numele *__VIEWSTATE*. Acesta este mecanismul prin care ASP.NET menţine starea unei pagini de la o trimitere la alta către server. Prima dată când se cere pagina, ASP.NET va codifica valoarea tuturor controalelor şi le va stoca în acest input ascuns. La fiecare cerere următoare a paginii acest input va fi folosit pentru a reiniţializa controalele Web şi HTML. În felul acesta este menţinută starea unor controale (de exemplu conţinutul unui input de tip text sau de tip select, pentru care s-a făcut alegerea unei opţiuni).

11.1.3 Code-behind

Se recomandă evitarea scrierii codului inline, deoarece asta duce la amestecarea părţii de design al paginii cu procesarea evenimentelor; în plus, este posibil ca procesarea să fie extrem de consistentă iar codul rezultat devine greu de urmărit şi menţinut. Erorile de sintaxă care se fac în codul inline sunt observabile doar la runtime, precompilarea fiind în acest caz imposibilă.

ASP.NET are o modalitate extrem de simplă pentru separarea părţii de interfaţă utilizator de cea de cod propriu-zis. E utilă separarea între cele două

aspecte, deoarece de partea de interfaţă se poate ocupa un web-designer, iar de cea programativă cineva cu cunoştinţe de .NET. Codul executat se va scrie într-un fişier cu extensia aspx.cs sau cs (dacă e vorba de cod C#) care va fi compilat sub forma unui fişier dll depus în directorul bin al aplicaţiei Web.

Pentru a se introduce partea de code-behind pentru pagina anterioară, se şterge scriptul inline şi se modifică astfel:

```
<%@ Page Inherits="Test1.HelloWorldCB" %>
<html>
  <head>
  </head>
  <body>
    <form method= post runat= server >
      <asp:Label id="lblHelloWorld" text="Hello World" runat="Server" />
      <br>
      <asp:Button onclick="ClickedIt" text="Submit" runat="Server" />
    </form>
  </body>
</html>
```

Pe primul rând s-a scris o directivă ASP.NET (ele pot fi scrise oriunde, dar se obiŋuieşte trecerea lor la începutul documentului). Directiva Page precizează în acest caz care este clasa din care se derivează pagina curentă. Clasa de bază se poate declara aşa:

```
namespace Test1
{
  using System;
  using System.Web;
  using System.Web.UI;
  using System.Web.UI.WebControls;
  public class HelloWorldCB: System.Web.UI.Page
  {
    protected Label lblHelloWorld;
    protected void ClickedIt(object sender, System.EventArgs e)
    {
      lblHelloWorld.Text = "Text schimbat" ;
    }
  }
}
```

Pagina aspx rezultată va moşteni clasa Test1.HelloWorldCB.

11.2 Clasa *Page*

Clasa *Page* este moştenită direct sau indirect de orice pagină ASP.NET; ea este responsabilă de asigurarea unei funcţionalităţi de bază pentru procesarea de pagini pe server.

Ciclul de viaţă a unei pagin ASP.NET este următorul:

1. Pagina este iniţializată. Aceasta include crearea instanţelor controalelor şi setarea event-handlerelor.
2. Partea de viewstate este procesată, populându-se controalele cu date.
3. Evenimentul de Load este apelat. In event-handlerul pentru acest eveniment se începe programarea logicii paginii.
4. Event-handlerele pentru controale sunt apelate (apăsări de butoane, schimbarea selecţiei curente dintr-un dropdown list, etc)
5. Se salvează viewstate
6. Pagina este tradusă în HTML (randare)

Dăm mai jos cele mai importante evenimente din clasa *Page*.

11.2.1 Evenimentul *Init*

Acest eveniment este primul apelat în ciclul de viaţă al unei pagini. În interiorul event-handlerului se creează toate controalele de pe pagină şi de asemenea se face asignează event-handlerele pentru acestea.

11.2.2 Evenimentul *Load*

Se apelează după *Init*. Are acces la partea de viewstate a paginii (inacceibilă pentru *Init*). Această metodă poate decide dacă este cerută de către client pentru prima dată sau este rezultatul unui postback, pe baza proprietăţii boolene *IsPostBack)*. Schiţa unei metode Load ar fi:

```
private void Page_Load(object sender, System.EventArgs e)
{
//cod care se executa la fiecare cerere
  if(!IsPostBack)
  {
    //cod care trebuie executat doar la prima cerere a paginii
    //ex: aducerea unor date din baza si popularea unor controale
```

```
  }
  else
  {
   //cod care se executa doar daca s-a facut postback
   //ex: verificarea unor intrari de pe forma
  }
}
```

11.2.3 Evenimentul *Unload*

Este apelat la descărcarea paginii din memoria serverului. Aici se dealocă resurse precum conexiuni la baza de date.

11.3 Crearea unei pagini ASP.NET folosind Visual Studio.NET

Deşi se poate scrie ASP.NET şi "de mână", este de preferat folosirea VS.NET pentru că permite crearea simplă a fişierelor aspx şi a celor de code-behind. În plus, se creează nişte fişiere de setări care ar fi bine să se găsească în folderul aplicaţiei.

Fişierele care se creează automat sunt Web.config, Global.asax, Global.asax.cs. Ultimul este util pentru cazul în care se doreşte procesarea evenimentelor de pornire a aplicaţiei sau de creare a sesiunii utilizator.

Fişierul Web.config conţine setări (de securitate, legate de sesiune) specifice aplicaţiei.

De asemenea s-a creat şi un web form (alt nume pentru pagină ASP.NET) numit implicit Webform1.aspx; din partea de Solution Explorer se poate modifica numele lui cu unul adecvat, de exemplu HelloWorld.aspx (se foloseşte această denumire în cele ce urmează).

Pe prima linie a fişierului HelloWorld.aspx apare:

```
<%@ Page language="c#" Codebehind="HelloWorld.aspx.cs"
AutoEventWireup="false" Inherits="HelloWorld.WebForm1"%>
```

Pentru directiva Page s-au specificat următoarele atribute:

- *language* cu valoarea *c#*; este limbajul în care se scrie în toate blocurile de cod de tip server;
- *Codebehind* specifică numele fişierului sursă; acesta se creează automat de către VS.NET;

- *Inherits* setează clasa care este moştenită de forma web; această clasă trebuie să fie derivată direct sau indirect din clasa *System.Web.UI.Page*.

Se trage de pe toolbox un control Label pe forma ASP.NET, iar în partea de proprietăţi se setează pentru ID valoarea lblHelloWorld, iar pentru text valoarea "Hello World." Codul generat de designer va fi:

```
<%@ Page language="c#" Codebehind="HelloWorld.aspx.cs"
AutoEventWireup="false" Inherits="HelloWorld.WebForm1" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<HTML>
 <HEAD>
  <title>WebForm1</title>
   <meta name="GENERATOR" Content="Microsoft Visual Studio .NET 7.1">
   <meta name="CODE_LANGUAGE" Content="C#">
   <meta name="vs_defaultClientScript" content="JavaScript">
   <meta name="vs_targetSchema"
   content="http://schemas.microsoft.com/intellisense/ie5">
  </HEAD>
  <body MS_POSITIONING="GridLayout">
   <form id="Form1" method="post" runat="server">
    <asp:Label id="lblHelloWorld" style="Z-INDEX: 101; LEFT: 32px;
    POSITION: absolute; TOP: 24px" runat="server">Hello World
    </asp:Label>
  </form>
 </body>
</HTML>
```

Se trage apoi un control de tip buton pe formă pentru care ID-ul implicit este Button1 iar textul va fi setat la "Buton"; pe partea de HTML a formularului codul va fi:

```
<form id="Form1" method="post" runat="server">
 <asp:Label id="lblHelloWorld" style="Z-INDEX: 101; LEFT: 32px;
 POSITION: absolute; TOP: 24px" runat="server">Hello World</asp:Label>
 <asp:Button id="Button1" style="Z-INDEX: 102; LEFT: 128px;
 POSITION: absolute; TOP: 24px" runat="server" Text="Buton"></asp:Button>
</form>
```

Dacă în toolbox, la partea de evenimente ataşate butonului se dă dublu click pe evenimentul *Click* în partea de code-behind se va genera automat metoda:

```
private void Button1_Click(object sender, System.EventArgs e)
```

```
{

}
```

atunci când butonul este apăsat se face o submitere a formularului către server şi codul conţinut în metoda de mai sus va fi executat. Codul va fi:

```
private void Button1_Click(object sender, System.EventArgs e)
{
  lblHelloWorld.Text = "Text schimbat";
}
```

La apăsarea butonului textul din componenta lblHelloWorld se va schimba, iar pagina nouă va fi trimisă către browser. Reîmprospătări succesive ale paginii (refresh) vor duce la afişarea aceluiaşi "Text schimbat", deoarece viewstate-ul conţine valoarea modificată de apăsarea butonului.

11.4 Controale server

11.4.1 Postback

Postback apare de câte ori se transmite pagina de la client înapoi la server. Există două modalităţi de realizarea a acestei trimiteri înapoi: prin folosirea unor elemente de tip `<input type="submit">` sau `<input type="image">` respectiv prin script executat pe client. De exemplu, dacă se trage pe formă un buton, codul generat de designer va fi:

```
<form method="post" runat="server" ID="Form1">
  <asp:Button id="Button1" runat="server" Text="Button 1"></asp:Button>
</form>
```

iar pe browser se va genera:

```
<form name="Form1" method="post" action="MyASPNETPage.aspx" id="Form1">
  <input type="submit" name="Button1" value="Button 1" id="Button1" />
</form>
```

Dacă pentru un formular nu se specifică partea de action, aceasta implicit va fi aceeaşi pagină.

Trimiterea formularului prin cod client s-ar face astfel:

```
<script language="javascript">
  function SubmitTheForm()
```

```
  {
   document.forms["Form1"].submit();
  }
</script>
```

iar proprietatea *onclick* a unui contro de tip HTML se setează la valoarea *SubmitTheForm()*:

```
<INPUT style="Z-INDEX: 103; LEFT: 80px; POSITION: absolute;
TOP: 104px" type="button" value="Button" onclick="SubmitTheForm()">
```

În cazul controalelor Web de tip server a căror modificare de stare trebuie să ducă la postback (de exemplu la selectarea unui item dintr-un combobox), se setează proprietatea *AutoPostBack* pe *true*, se setează o metodă pentru evenimentul de schimbare a conţinutului (în cazul nostru: *SelectedIndexChanged*) iar pagina HTML generată pe client va conţine o funcţie javaScript de submitere a formularului:

```
function __doPostBack(eventTarget, eventArgument) {
  var theform;
  if (window.navigator.appName.toLowerCase().indexOf("microsoft") > -1) {
    theform = document.Form1;
  }
  else
  {
    theform = document.forms["Form1"];
  }
  theform.__EVENTTARGET.value = eventTarget.split("$").join(":");
  theform.__EVENTARGUMENT.value = eventArgument;
  theform.submit();
}
```

unde *__EVENTTARGET* şi *__EVENTARGUMENT* sunt două câmpuri ascunse. Combobox-ul arată astfel în codul HTML client:

```
<select name="DropDownList1" onchange="__doPostBack('DropDownList1','')"
language="javascript" id="DropDownList1" style="Z-INDEX: 104; LEFT:
312px; POSITION: absolute; TOP: 120px">
  <option value="1">opt 1</option>
  <option value="2">opt 2</option>
</select>
```

11.4.2 Data Binding

Data Binding permite legarea unei proprietăţi a unui control la o sursă de date, precum un câmp în baza de date. Această legare se poate face scriind expresii în interiorrul unei declaraţii de control în fişierul .aspx sau în code-behind.

Pentru a se lega direct în fişierul .aspx se foloseşte tag-ul `<%# %>` (expresie Data Binding). Această metodă se aplică pentru orice control care nu este listă precum Label, TextBox, CheckBox. Când Data Binding se face pentru un control de tip listă (DropDownList sau CheckBoxList) se va folosi proprietatea DataSource a controlului la o sursă de date oarecare (e.g. DataSet). Aceasta paote fi făcută în partea de code-behind sau într-un bloc de tip script din fişierul .aspx.

Toate controalele care au posibilităţi de Data Binding implementează o metodă numită DataBind(). Aceasta se ocupă de evaluarea expresiilor de biding în fişierul .aspx sau pentru sursele de date specificate în proprietatea DataSource. Este important de reţinut că expresiile Data Binding şi proprietăţile DataSource nu sunt evaluate până când nu se apelează metoda DataBind() ale controalelor cu care sunt asociate. Deoarece Page este el însuşi un control, are o implementare de metodă DataBind(), care când este apelată va duce la apelul tuturor metodelor DataBind() pentru controalele copil.

11.5 Controale Web

Controalele Web reprezintă o modalitate uşoară de a interaţiona cu serverul. Nu există o relaţie univocă între controale web şi elemente Html, ci controalele web vor randa pe parte de client cod Html specific browserului. Clasa Page are o proprietate nestatică numită *ClientTarget* care permite definirea browserului client. Valorile pentru proprietate sunt:

- Auto - se va face autodetectarea browserului
- DownLevel - se presupune că browserul ştie doar Html 3.2 sau mai puţin
- UpLevel - browserul ştie Html 4 sau mai mult, CSS, EcmaScript, Microsoft Document Object Model

Orice control Web este derivat din System.Web.UI.WebControls care este derivat din System.Web.UI.Control. Aceasta din urmă pune la dispoziţie partea de Data Binding, viewstate, evenimente precum Init, Load, Unload,

etc. Clasa WebControl conţine proprietăţi de stil precum bacgkround, height, width.

11.5.1 Label

Conţine text de tip read-only, asignat static sau prin Data Binding. Se generează pe browser ca fiind un element de tip *span*. Pentru crearea unei etichete cu text predefinit se trage un control Web de pe toolbar şi se pot seta proprietăţi ale lui, ceea ce ar putea genera următorul cod în designer:

```
<asp:Label id=Label1 runat="server" Width="70px" Height="50px">
  This is a label
</asp:Label>
```

Dacă se doreşte folosirea DataBinding, atunci se poate scrie:

```
<asp:Label id=Label1 style="Z-INDEX: 101; LEFT: 10px; POSITION:
  absolute; TOP: 10px" runat="server"
  text="<%# DateTime.Now.ToString() %>">
</asp:Label>
```

Pentru ca valoarea să fie efectiv calculată şi afişată pe parte de browser trebuie chemată metoda *DataBind()* pentru obiectul etichetă; acest lucru se poate face în partea de încărcare a paginii:

```
protected void Page_Load(object sender, System.EventArgs e)
{
  Label1.DataBind();
}
```

11.5.2 Button

Permite crearea unui buron de tip submit şi care poate să determine executarea unei metode C# pe parte de client, pe baza unui event handler definit în code-behind.

În cazul în care se doreşte crearea mai multor butoane care să fie procesate pe server, e nevoie de un mecanism care să facă diferenţierea lor, astfel încât să se poată determina care a fost butonul apăsat. Acest lucru se face prin proprietăţile *CommandName* şi *CommandArgument* de tip string. Event handlerul pentru butoane poate să fie:

```
private void Button_Command(object sender,
  System.Web.UI.WebControls.CommandEventArgs e)
```

```
{
  //This Is a Command event handler
}
```

unde obiectul *e* are proprietăţile *CommandName* şi *CommandArgument* care vor da valorile setate pentru controale.

11.5.3 LinkButton

Se comportă ca şi un buton; codul Html generat va fi de forma `<a>` iar la apăsare va apela funcţia Javascript `__doPostBack` prezentată mai sus.

11.5.4 TextBox

Dă posibilitatea de a introduce text într-un formular. Se poate folosi în trei moduri: *SingleLine*, *MultiLine* sau *Password*. Prezintă proprietăţi de tipul *Columns, Rows, Wrap, MaxLength, ReadOnly, Text, AutoPostBack* şi un eveniment *TextChanged*. Dacă proprietatea *AutoPostBack* este setată la true, pagina va fi automat retransmisă la server dacă se iese din controlul curent şi textul a fost schimbat; mecanismul se bazează pe acceaşi funţie Javascript ca la LinkButton.

11.5.5 CheckBox

Determină apariţia unui checkbox; proprietăţile de interes sunt *Checked, Text, TextAlign, AutoPostBack* iar evenimentul detectat este *CheckedChanged*.

11.5.6 RadioButton

Derivată din *CheckBox*, are în plus proprietatea *GroupName* care permite gruparea mai multor astfel de butoane.

11.5.7 DropDownList

Permite crearea unei liste de opţiuni stocate sub numele de nume şi valoare. Toate item-urile conţinute sunt accesibile via proprietatea *Items* de tip *ListItemCollection*. Fiecare element al unei astfel de coleţii este de tip *ListItem* şi are proprietăţile *Text*, *Value* de tip String şi *Selected* de tip boolean. Lista de opţiuni se poate crea fie prin introducere directă în proprietatea *Items*, fie prin Data Binding. Evenimentul care se pate detecta este *SelectedIndexChanged* şi se poate procesa pe server.

Curs 12

ASP.NET (2)

12.1 Controale de validare

Controalele de validare sunt utilizate pentru a face verificari asupra îndeplinirii unor condiţii pentru conţinutul controalelor de introducere a datelor. Această validare se poate face pe server sau (de preferat în majoritatea cazurilor) pe client (dacă acesta suportă DHTML). Validări suplimentare mai complexe pot apărea şi pe server, chiar dacă au fost efectuate validări primare pe client. Unui control de preluare a datelor i se pot asocia oricâte controale de validare. De exemplu, se poate folosi un control de tipul *RequiredFieldValidator* şi altul care să valideze conţinutul folosind expresii regulate; doar primul control poate determina dacă s-a introdus ceva, celelalte returnând automat valoare de adevăr dacă controlul de intrare nu conţine nimic.

Orice astfel de control de validare este derivat din clasa *BaseValidator*, care prezintă proprietatea *ControlToValidate* în care se depune ca valoare ID-ul controlului pentru care se face validarea. Proprietatea *Display* determină dacă se rezervă loc pe pagină pentru a se afişa mesajul de eroare. Poate avea valorile *Static* şi *Dynamic*: în primul caz se rezervă suficient spaţiu pentru mesajul de eroare, chiar dacă acesta nu este vizibil; în al doilea caz dacă mesajul de eroare este arătat, atunci pot apărea deplasări ale componentelor de pe pagină pentru a face loc textului de eroare. Mesajul de eroare se defineşte în proprietatea *ErrorMessage*.

Proprietatea *EnableClientScript* setată pe true va duce la generarea de cod scripting pentru client (JavaScript sau VBScript); dacă proprietatea este setată pe *false* se va face validarea doar pe server. *Trebuie reţinut faptul că validarea se face întotdeauna şi pe server.* Pentru un client de tip non-DHTML, pentru un asemenea control nu se va genera nici un cod scripting client, ci atunci când formularul este trimis către server se va face testare va-

lidităţii datelor introduse; dacă ceva nu corespunde criteriilor se va retrimite pagina către client împreună cu mesajele de eroare scrise.

În acest context spunem că clasa *Page* conţine o proprietate *IsValid* care va fi setată la *true* dacă toate validatoarele sunt îndeplinite pe parte de server sau *false* în caz contrar.

Controalele de validare sunt: *RequiredFieldValidator*, *RegularExpressionValidator*, *ValidationSummary*, *RangeValidator*, *CompareValidator*, *CustomValidator*.

12.1.1 RequiredFieldValidator

Forţează utilizatorul să introducă ceva într-un control de introducere. Dacă se încearcă submiterea formularului iar câmpul pentru care s-a setat vaidatorul nu are valoare introdusă se va semnala eroare prin apariţia mesajului definit de utilizator în proprietatea *ErrorMessage*. Codul JavaScript care se efectuează aceste validări este scris în fişierul sursă C:\Inetpub\wwwroot\aspnet_client\system_web\1_1_4322\WebUIValidation.js.

12.1.2 RegularExpressionValidator

Verifică pe baza unei expresii regulate dacă valoarea introdusă într-un câmp este corectă. Pe parte de client se va folosi sintaxă de regex-uri de tip JavaScript, pe parte de server se va folosi clasa *Regex*[1]. În Visual Studio sunt definite câteva expresii regulate care se pot aplica pentru diferite scopuri: adrese de email, numere de telefoane din diverse ţări, URL, etc.

12.1.3 RangeValidator

Verifică dacă valoarea dintr-o intrare este cuprinsă între un minim şi un maxim. Pentru conţinut şi extreme tipul poate fi *String, Integer, Double, Date* şi *Currency*, specificat în proprietatea *Type* a controlului. Valorile minime şi maxime admise se dau în proprietăţile *MinimumValue* şi *MaximumValue*.

12.1.4 CompareValidator

Se foloseşte pentru a compara conţinutul unui control cu conţinutul altui control sau cu o valoare fixă, predefinită. Dacă se doreşte compararea cu

[1]Pentru o coleţie bogată de expresii regulate, a se vedea şi www.regexlib.com

valoarea dintr-un alt control, se setează adecvat proprietatea *ControlToCompare*, iar dacă se face compararea cu o valoare predefinită atunci se setează proprietatea *ValueToCompare*. De asemenea se poate specifica tipul de dată pentru care se face compararea, ca la *RangeValidator*. Proprietatea *Operator* specifică operatorul folosit la comparare: *Equal* (implicit), *NotEqual, GreaterThan, GreaterThanEqual, LessThan, LessThanEqual*, şi *DataTypeCheck* (operator unar, care verifică validitatea datei calendaristice specificate).

12.1.5 CustomValidator

Permite crearea de validatoare de către programator, care se pot executa atât pe client cât şi pe server. Pentru validarea pe server trebuie dat un delegat de tipul:

```
void HandlerName (object source, ServerValidateEventArgs args)
```

Al doilea parametru are proprietatea *Value* care dă acces pe server la valoarea introdusă pe client şi proprietatea *IsValid* de tip bool care trebuie setată pe *true* dacă validarea s-a făcut cu succes sau *false* în caz contrar. Această valoare afectează valoarea proprietăţii *IsValid* a obiectului de tip *Page*.

Motivul pentru care s-ar folosi un asemenea validator ar fi că verificarea se poate face numai pe server (de exemplu se verifica dacă numele si parola unui utilizator se regasesc in baza de date). Dacă însă se doreşte validarea şi pe client folosind JavaScript, atunci se va seta în mod adecvat proprietatea *ClientValidationFunction*.

Exemplu: pentru validarea pe server a valorii introduse într-un câmp, se poate scrie codul C#:

```
private void vldUserIDCstm_ServerValidate(object source,
   ServerValidateEventArgs args)
{
  if (args.Value == "JoeHealy" || args.Value == "SpikePierson" ||
    args.Value == "AndyJohnston")
  {
    args.IsValid = false;
  }
  else
  {
    args.IsValid = true;
  }
}
```

Pentru a se face aceeaşi verificare şi pe client, se poate scrie codul următor la sfârşitul paginii aspx:

```
<script language="JavaScript">
function vldUserIDCstm_ClientValidate(source, args)
{
  if (args.Value = "JoeHealy" || args.Value = "SpikePierson" ||
  args.Value = "AndyJohnston")
  {
    args.IsValid=false;
  }
  else
  {
    args.IsValid=true;
  }
}
</script>
```

Numele funcţiei JavaScript este setat în proprietatea *ClientValidationFunction*.

12.1.6 ValidationSummary

Nu este un control de validare propriu zis, fiind derivat din clasa *WebControl*. Este folosit pentru a afişa mesajele de eroare pentru toate controalele de validare într-un singur loc. Proprietatea *DisplayMode* este folosită pentru a controla formatul în care se vor afişa mesajele de eroare, având valorile posibile: *List*, *BulletList*, *SingleParagraph*. De asemenea există proprietatea *HeaderText* care permite afişarea unui antet deasupra listei de erori.

Mesajele de eroare pot fi afişate într-unul din 2 moduri sau în ambele. Ele pot fi afişate în pagină de orice browser, fie el *UpLevel* fie *DownLevel*. În plus, un browser *UpLevel* poate să afişeze mesajul (şi) într-o fereastră de eroare creată de browser. Dacă proprietatea de tip bool *ShowSummary* este setată pe true, atunci se vor afişa mesajele de eroare în pagina HTML. Dacă proprietatea de tip bool *ShowMessageBox* este pusă pe true, atunci va apărea o fereastră cu lista de erori; în acest din urmă caz, pentru un browser de tip DownLevel mesajul nu va fi arătat în fereastră.

12.2 Comunicarea cu browserul

Pentru trimiterea de date către client se poate folosi pe parte de server obiectul predefinit *Response* de tip *HttpResponse*. Deşi ASP.NET pune la

dispoziţie mecanisme suficient de complexe pentru a nu necesita folosiera masivă a acestui mecanism, *Response* este util pentru:

1. pagini foarte simple sau debugging;
2. modificarea mecanismului de buffering;
3. redirectarea clientului;
4. managementul mecanismului de caching

12.2.1 Generarea programatică conţinutului

În unele cazuri răspunsul care se trimite de la server la browserul client poate fi suficient de simplu pentru a nu necesita utilizarea de controale. Utilitatea unei asemenea metode este limitată, dar există situaţii în care calea se dovedeşte a fi bună.

Exemplu:

```
protected void Page_Load(object sender, System.EventArgs e)
{
  Response.Write(string.Format("The date is: {0}<br>",
  DateTime.Now.ToShortDateString()));
  Response.Write(string.Format("<i>The time is: <b>{0}</b></i><br>",
  DateTime.Now.ToShortTimeString()));
}
```

De asemenea se permite trimiterea conţinutului unor fişiere căte browser, via metoda *WriteFile*.

12.2.2 Redirectarea

Se foloseşte pentru cazul în care pagina care se trimite clientului nu este cea curentă, ci o alta. De exemplu, pentru cazul în care se doreşte redirectarea utilizatorului căter o pagină de login. Redirectarea trebuie făcută înainte ca vreun header să fie trimis către client.

Exemplu:

```
Response.Redirect("http://mysite.com/myBusiness");
```

Această redirectare are loc de pe server, fără a se trimite cod către browser care să îl determine pe acesta să ceară o altă resursă.

12.3 Cookies

Cookie-urile reprezintă informaţie stocată pe client. Deşi nu reprezintă un mecanism garantat pentru lucrul cu clientul, în unele cazuri pot fi suficient de utile pentru o aplicaţie. Există 4 clase de lucru cu cookie-urile:

- *HttpCookie*
- *HttpCookieCollection*
- *HttpResponse*
- *HttpRequest*

Obiectul *Response* are o colecţie numită *Cookies*. Când un browser face o cerere către un server, va trimite prin intermediul headerelor toată colecţia de cookie-uri care aparţin acelui site. Această colecţie se încarcă în *Cookies*.

Atunci când se trimite un cookie de pe server pe client se poate preciza dacă acesta va fi valabil doar pe perioada cât este deschis browserul (aşa-numitele cookie-uri *de sesiune*) sau şi după ce browserul este închis (cookie-uri *persistente* - rezistă până la o dată a expirării setată sau pentru totdeauna). Primul tip de cookie-uri sunt utile în medii în care nu există intimitate a utilizatorului, un cont fiind folosit de către mai mulţi; al doilea tip se poate folosi pentru stocarea informaţiilor (precum preferinţele) de la o vizită la alta.

Diferenţa dintre cele două din punct de vedere programatic constă în absenţa sau prezenţa unui timp de expirare. Cele de tip sesiune nu au setat nici un timp de expirare (şi se reţin în memoria RAM a calculatorului), pe când celelalte pot avea un timp setat în viitor (fiind persistate pe disc).

Exemplu de creare a unui cookie de tip sesiune:

```
protected void Page_Load(object sender, System.EventArgs e)
{
  HttpCookie tempcookie = new HttpCookie("myCookie");
  tempcookie.Values.Add("userid", "1250");
  Response.Cookies.Add(tempcookie);
}
```

Pentru crearea unui cookie persistent, se poate proceda ca mai jos:

```
protected void Page_Load(object sender, System.EventArgs e)
{
  HttpCookie perscookie = new HttpCookie("myCookie");
  perscookie.Expires = Convert.ToDateTime("24/05/05 16:00");
```

```
  perscookie.Values.Add("userid", "1250");
  Response.Cookies.Add(perscookie);
}
```

Accesarea acestui cookie se face pe server prin intermediul obiectului *Request*:

```
HttpCookie cookie = Request.Cookies["myCookie"];
if (cookie != null)
{
  string s = cookie.Values["userid"].ToString();
  ...
}
```

12.4 Fişierul de configurare al aplicaţiei Web

Diverse setări ale aplicaţiei pot fi menţinute într-un fişier de tip XML numit *Web.config*. Structura şi modul de accesare a conţinutului acestuia este asemănătoare cu cea prezentată în secţiunea 8.4.4. Deosebit este însă faptul pentru o aplicaţie Web această configurare respectă o anumită ierarhie. La rădăcina acestei ierarhii se află fişierul *machine.config* aflat în diorectorul %windir%\Microsoft.NET\Framework\Version\CONFIG. Setările de aici pot fi suprascrise de setări aflate în directoarele *Web.config* unei aplicaţii ASP.NET.

12.5 Fişierul *global.asax*

Este un fişier opţional structurat XML care prezintă setări relative la aplicaţia ASP.NET (mai exact, pentru obiectul reprezentând aplicaţia). Fişierul se găseşte în rădăcina aplicaţiei, este compilat ca o clasă care este derivată din *HttpApplication*, fiind recompilată automat la fiecare modificare a sa.

Metodele care se pot implementa în această clasă sunt:

- *Application_Start* - apelată atunci când un client cere pentru prima oară o pagină de la această aplicaţie; folosită pentru iniţializarea unor resurse (grupuri de conexiuni, valori din regiştri, etc care sunt accesate de către părţi din aplicaţie).
- *Session_Start* - ori de câte ori se porneşte o sesiune
- *Application_BeginRequest* - apelat ori de câte ori un client cere o pagină oarecare de la server; s-ar putea folosi la contorizarea numărului de accese la pagină

- *Application_EndRequest* - apelat la sfârşitul oricărei cereri către server
- *Application_AuthenticateRequest* - apelată la fiecare cerere de autentificare a unui utilizator
- *Application_Error* - apelată ori de câte ori apare o eroare care nu a fost procesată
- *Session_End* - la terminarea unei sesiuni prin expirare
- *Application_End* - apare la o resetare a serverului sau atunci când se recompilează aplicaţia ASP.NET.

12.6 Managementul sesiunii

Sesiunea este un mecansim prin care se creează impresia unei conexiuni permanente de la client la server. O sesiune este stocată pe server şi va reţine toate obiectele necesare bunei funcţionări pe perioada respectivă (de exemplu poate conţine coşul de cumpărături pentru un client, ID-ul acelui client, etc). Ori de câte ori un client cere prima pagină a unei aplicaţii Web, se creează un obiect de sesiune pe server identificat printr-un număr unic care este de asemenea trimis clientului printr-un cookie. Această sesiune este ataşată doar browserului care a cerut crearea ei; dacă acesta se închide, atunci vechea sesiune este inaccesibilă. De fiecare dată când se cere o pagină de la server (după ce sesiunea a fost creată), se cere id-ul stocat în cookie-ul de pe client şi se află care este sesiunea ataşată.

Stocarea unei valori în sesiune trebuie să se facă folosind o cheie (unică pe acea sesiune) şi o valoare ataşată (de ori ce tip). Exemplu:

```
Session["username"] = "jsmiley";
```

Dacă cheia *username* nu există, se va crea şi i se va ataşa string-ul numit; dacă există, valoarea sa se va suprascrie. Recuperarea valorii anterioare se face cu:

```
String uname = (string)Session["username"];
```

Menţionăm că în sesiune se pot stoca orice obiecte, nu doar şiruri de caractere. Dacă serverul web se reporneşte, toate valorile din acea sesiune se pierd; există alternativa de a stoca sesiunile pe un alt server sau chiar pe un server SQL. Ştergerea unei chei şi a valori ataşate din sesiune se face folosind metodele *Session.Remove(String key)*, *Session.RemoveAt(int index)*. Golirea sesiunii se face cu *Session.RemoveAll()* sau *Session.Clear()*.

12.7 ViewState

ViewState reprezintă o colecţie de valori care se stochează pe client, dar nu prin intermediul cookie-urilor, ci pe baza unui câmp de tip hidden din pagina HTML trimisă clientului. Este folosită în special pentru a reţine valorile controalelor Web de-a lungul mai multor post-back-uri.

Spre deosebire de obiectul *Session* care poate stoca orice tip de obiect, obiectul ViewState (de tip *StateBag*) poate să conţină doar string-uri.

Exemplu:

```
ViewState["uname"] = "Joe";
..
String suname = (string)ViewState["uname"];
```

Se poate şterge selectiv din obiectul *ViewState* prin metoda *Remove(String cheie)*.

12.7.1 Application

Obiectul *Application* este foarte asemănător cu *Session*, cu deosebirea că nu este unic pentru fiecare sesiune utilizator, ci pentru întreag aplicaţie. Este utilizat pentru menţinearea unor obiecte cu caracter global, utile pentru orice sesiune (valori din regiştri sau valori imuabile care nu trebuie citite de fiecare dată). Stocarea de valori se face prin:

```
Application["x"] = new MyObject();
```

iar accesarea lui prin:

```
MyObject x = Application["x"];
```

12.8 Alte puncte de interes în aplicaţii ASP.NET

- crearea controalelor utilizator - a paginilor care sunt convertite în controale pentru a permite reutilizarea lor mai simplă - ex: header, footer;

- separarea codului de prezentare - pentru a permite separarea muncii între web designer şi programator; separarea pe mai multe nivele uşurează menţinerea codului şi schimbarea lui; este sugerată folosirea a cel puţin 3 nivele: interfaţa utilizator, nivelul de business logic, nivelul de acces la date;

- crearea de aplicaţii web pentru dispozitive mobile; deşi la ora actuală piaţa de dispozitive mobile este relativ redusă, există totuşi un segment bine definit care cere astfel de aplicaţii. Design-ul specific şi capabilităţile unui astfel de browser pot ridica probleme noi;
- politici de caching - pentru optimizarea accesului la resurse;
- autentificarea - diverse forme de securitate: formulare, .NET Passport;
- folosirea protocoalelor de comunicaţie securizate (SSL)
- folosirea serviciilor Web - servicii care se pot accesa în Internet; bazate pe XML şi protocoale de comunicaţie larg răspândite, un serviciu Web este independent de platformă.

Curs 13

Servicii Web

13.1 Generalităţi

Un serviciu Web (SW) expune o interfaţă de invocare a unei activităţi de către un client. Un client poate accesa un serviciu Web folosind standarde Internet. Un serviciu web rezolvă următoarele probleme:

- *interoperabilitate* - un SW trebuie să permită comunicarea între orice platforme;
- *interfeţe puternic tipizate* - nu trebuie să existe ambiguitate relativ la tipul de date trimise sau primite spre/de la un SW. Aceste tipuri de date ar trebui să se poată mapa rezonabil peste tipurile de date folosite de limbajele procedurale existente;
- *folosirea standardelor Internet* - implementarea unui SW trebuie să se bazeze pe standardele şi protocoalele de comunicare deja existente, pentru a evita "reinventarea roţii"
- *acces multilimbaj* - un SW nu trebuie să reducă aria limbajelor care le pot utiliza; transparenţa trebuie să se manifeste atât la nivelul limbajului în care s-a creat SW, cât şi la nivelul clientului;
- *posibilitate de integrare în orice infrastructură distribuită* - un SW nu trebuie să fie strâns cuplat de o anumită infrastructură de comunicare.

13.1.1 Componentele unui WS

Principalele blocuri componente ale unui serviciu Web sunt redate în figura 13.1.

- *Descoperirea* - dacă se doreşte conectarea la un serviciu Web, este necesară o modalitate de a determina locaţia serviciului - proces numit descoperire. Descoperirea poate fi realizată prin intemediul unor directoare centralizate sau prin alte metode ad-hoc.
- *Descrierea* - după ce s-a rezolvat problema de la punctul precedent, clientul va avea nevoie de o descriere a modului de interacţiune cu SW. Descrierea reprezintă metadate structurate despre interfaţa ce urmează a fi "consumată", precum şi documentaţie scrisă despre SW conţinând detalii sau exemple de utilizare.
- *Formatul mesajelor* - necesar a fi precizat pentru a se putea codifica într-un mod convenabil toată conversaţia între client şi SW. Acest lucru permite abstractizarea protocoalelor de comunicare şi concentrarea pe logica problemei.
- *Codificarea* - datele trimise între cei 2 parteneri de conversaţie trebuie codificate într-un mod care să permită procesarea lor de către orice limbaj. XML este alegerea potrivită pentru aşa ceva, fiind structurat şi de tip text.
- *Transportul* - modul prin care datele serializate care compun un mesaj sunt trimise de la un capăt la celălalt. S-a cerut explicit funcţionarea unui serviciu web peste protocoale "prietenoase", care să nu necesite setări speciale de securitate.

13.2 Discuţii asupra blocurilor componente

13.2.1 Protocolul de transport

O aplicaţie bazată pe SW poate să fie folosită atât în LAN cât şi într-o reţea de mare întindere. Protocoale precum HTTP sau SMTP sunt testate din punct de vedere al serviciilor pe care îl oferă. Faptul că HTTP este fără stare permite crearea unor aplicaţii care să permită continuarea servirii unor clienţi chiar dacă serverul iniţial a căzut (sarcina sa fiind preluată de alte servere). În plus, SMTP şi HTTP sunt portocoale care nu cer o configurare specială a firewall-urilor care delimitează o reţea.

13.2.2 Schema de codificare

XML reprezintă alegerea naturală pentru codificarea mesajelor dintre client şi server, deoarece este structurat (făcând procesarea sa uşoară) şi

Figura 13.1: Componentele unui serviciu Web

în acelaşi timp neutru din punct de vedere al platformei. În acest fel dispar problemele legate de modul de reprezentare a datelor pe diverse arhitecturi de sisteme: textul poate fi procesat la fel de către oricine. În plus, există o multitudine de parsere XML care pot fi utilizate chiar şi pe platforme cu putere de procesare redusă.

13.2.3 Convenţia de formatare

Alături de corpul mesajului se transmit de cele mai multe ori şi metadate. *Simple Object Access Protocol* (SOAP) este un protocol care descrie regulile şi formatul pentru comunicare. Prin intermediul lui SOAP se pot transmite documente şi se pot face apeluri de metode:

```
<soap:Envelope
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <PurchaseOrder ID="3245">
```

```
     <From>Jim</From>
     <To>Frank</To>
     <Items>
          <Item ID="111">Dingbats</Item>
          <Item ID="222">Widgets</Item>
     </Items>
    </PurchaseOrder>
  </soap:Body>
</soap:Envelope>
```

În exemplul de mai sus se transmite conţinutul unei comenzi de la Jim la Frank. Pentru apelul unei metode la distanţă s-ar putea folosi:

```
<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body>
       <m:GetStockPrice xmlns:m="http://contoso.com/Stock">
           <symbol>MSFT</symbol>
       </m:GetStockPrice>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

corespunzător apelului de metodă C#:

```
GetStockPrice( "msft" );
```

Structura generală a unui mesaj SOAP este un "plic"[1] care este format din header-e (metadate asociate mesajului) şi conţinut propriu-zis.

13.2.4 Mecanismul de descriere

Web Services Description Language (WSDL) defineşte un document XML care permite descrierea serviciului Web ţintă. Pemite descrierea conţinutului mesajului dintre client şi serviciu, realizarea de tipuri de date utilizator, formatul de comunicare între client şi server.

13.2.5 Mecanisme de descoperire

Universal Description, Discovery, and Integration (UDDI) este mecansimul prin care se aduce la cunoştinţă existenţa unor servicii web disponibile

[1] Engl: envelope

oricărui client. UDDI este un director centralizat care poate fi folosit pentru anunţarea de SW, pentru care se poate face căutarea după criterii: categoria, tipul de SW, compania, etc. De asemenea se poate folosi şi DISCO, un format XML proprietar Microsoft, având ca principal consumator Visual Studio .NET.

13.3 Crearea unui serviciu Web simplu

Mai jos exemplificăm crearea unui SW şi a unor clienţi care să interacţioneze cu acestea.

13.3.1 Adunarea a două numere

Vom crea un serviciu Web care preia două numere întregi şi returnează suma lor. Manual, acest lucru se face prin crearea unui director virtual (de ex. Test), se creează un fişier cu extensia asmx în acest director (fie el Test.asmx) şi se scrie codul:

```
<%@ WebService Class="TestClass" Language="C#" %>

using System;
using System.Web.Services;

public class TestClass {

     [WebMethod]
     public int Add( int a, int b)
     {
          return a+b;
     }
}
```

Drept client se poate folosi browser-ul IE cu adresa: `http://localhost/Test/Test.asmx`. Răspunsul este un document XML:

```
<?xml version="1.0" encoding="utf-8" ?>
  <int xmlns="http://tempuri.org/">5</int>
```

Descrierea pentru acest serviciu (documentul WSDL) este obţinută prin adresa: `http://localhost/Test/Test.asmx?WSDL`.

13.3.2 Validarea unui număr de card de credit

Să presupunem că avem de implementat un site de tip B2C (de exemplu vânzare de cărţi). Atunci când se efectuează plata prin card de credit se cere de la utilizator numărul de card şi data expirării. Se presupune că această validare nu este făcută de către aplicaţie, ci pe baza unui serviciu Web specializat. Vom exemplifica cele ce urmează prin Visual Studio, deşi se poate folosi şi altă modalitate.

Pentru Web Service se poate proceda astfel: File->New->Project->se alege ASP.NET Web Service->se completează numele PlataWS. Din Solution Explorer se modifică numele fişierului în PlataWS.asmx. În partea de cod se scrie metoda:

```
[WebMethod]
public bool ValidateCardNo(String cardNo, String date)
{
  return int.Parse(cardNo) %2 == 0;
}
```

(o validare simplă, pe baza parităţii numărului de card).

Pentru partea de site se poate crea o formă windows de tipul celei din figura 13.2, în care să se poată introduce numărul de card şi data expirării.

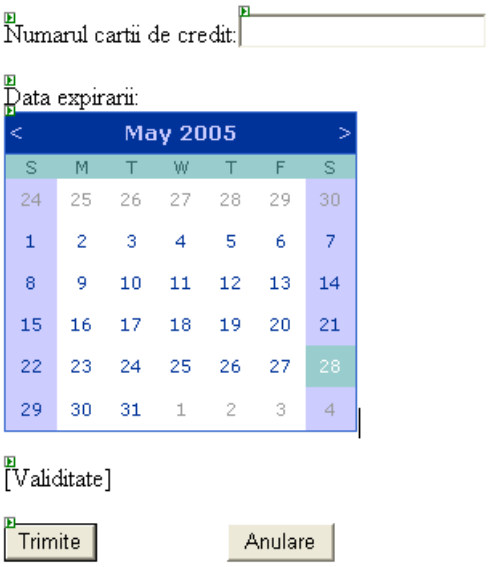

Figura 13.2: Preluarea datelor unei cărţi de credit

Pentru a se putea accesa serviciul web, se intră în Solution Explorer şi se adaugă la References un Web Reference la serviciul local PlataWS numit

localhost. Aceasta permite crearea unui obiect de tipul PlataWS pentru care se va apela metoda de validare. Codul apelat pe server la apăsarea butonului de submitere este:

```
private void Button1_Click(object sender, System.EventArgs e)
{
  localhost.PlataWS validator = new localhost.PlataWS();
  if (!validator.ValidateCardNo(NrCarte.Text,
    Calendar1.SelectedDate.ToString()))
  {
    Validitate.Text = "Invalid";
    Validitate.Visible = true;
  }
  else
  {
    Validitate.Visible = false;
  }
}
```

unde Validitate este un Label invizibil de pe formă. Validitatea este deci testată prin intermediul unui serviciu Web care returnează *true* sau *false*, în funcţie de care se afişează sau nu eroare.

13.3.3 SW pentru transmitere de conţinut binar

Folosind SOAP se poate transmite orice fel de mesaj, indiferent de sursa de provenienţă a acestuia. Vom arăta mai jos modul în care se poate face transmiterea unui fişier binar folosind servicii Web[2].

Pentru SW se va scrie codul:

```
[WebMethod]
public FileContent GetFile(String name)
{
  FileStream fileStream = File.OpenRead(name);
  FileContent fileContent = new FileContent();
  fileContent.Name = name;
  fileContent.Content = new byte[fileStream.Length];
  fileStream.Read(fileContent.Content, 0, fileContent.Content.Length);
  fileStream.Close();
  return fileContent;
}
```

[2]Cu toate că FTP este mai potrivit pentru aşa ceva

unde *FileContent* este o structură definită în acelaşi spaţu de nume:

```
namespace FileTransfer
{
public struct FileContent
{
public string Name;
public byte[] Content;
}
}
```

Aplicaţia client poate să fie de tip formă Windows, ce permite specificarea căii absolute a unui fişier aflat pe server şi care salvează conţinutul obiectului de tip *FileTransfer* pe discul local. Pentru accesarea serviciului Web se adaugă o referinţă Web la serviciul creat anterior (denumită mai jos *site*), astfel putând-se referi orice tip de date declarat public în interiorul spaţiului de nume. Secvenţa de cod pentru apelarea serviciului Web este:

```
private void button1_Click(object sender, System.EventArgs e)
{
  site.FileTransfer transfer = new site.FileTransfer();
  site.FileContent fileContent = transfer.GetFile(textBox1.Text);
  Stream sw = File.OpenWrite(@"c:\temp\output.bin");
  sw.Write( fileContent.Content, 0, fileContent.Content.Length );
  sw.Close();
}
```

13.4 SOAP

SOAP (Simple Object Access Protocol) specifică modul de împachetare a mesajelor. Câteva avantaje ale SOAP-ului sunt:

- nu este puternic cuplat cu vreun limbaj de programare, nu se specifică un API. SOAP se poate folosi din orice limbaj sau de pe orice platformă
- nu este legat de vreaun protocol de transport anume. Deoarece un mesaj SOAP etse un document XML, pentru el se poate folosi orice fel de protocol care ştie să transmită text
- nu este legat de nici o infrastructură distribuită (DCOM, Corba, etc)
- se foloseşte de standarde existente - XML, XML schema, HTTP, SMTP, etc

- interoperabilitate între platforme: SOAP poate fi folosit de către PC-uri, dispozitive mobile, mainframe-uri, etc

Un mesaj SOAP constă într-un plic compus în principal din header-e şi corp.

13.4.1 Header-e

Elementele de header sunt folosite pentru a transmite conţinut care nu are legătură propriu-zisă cu corpul mesajului. De exemplu, dacă se foloseşte un anumit algoritm de compresie a conţinutului mesajului atunci receptorul trebuie să ştie care este acesta. De asemenea se pot include:

- autentificare - se poate cere receptorului să se autentifice înainte de a fi procesat mesajul
- informaţie de tip rezumat sau semnătură digitală pentru a se asigura că conţinutul mesajului nu afost alterat
- informaţie de rutare - dacă mesajul trebuie să ajungă la mai multe destinaţii

Atributul *mustUnderstand*

Headerele pot fi ignorate de către receptor. Dar pentru o parte a acesora se poate cere ca să se asigure înţelegerea unui anumit header să fie garantată de către client sau ca întregul mesaj să fie ignorat. Într-un asemenea caz se specifică pentru acel element de header atributul *mustUnderstand* cu valoarea 1.

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Header>
<TransactionId soap:mustUnderstand="1">123</TransactionId>
</soap:Header>
<soap:Body>
<UpdateAccountInfo>
<email>sshort@microsoft.com</email>
<firstName>Scott</firstName>
<lastName>Short</lastName>
</UpdateAccountInfo>
</soap:Body>
</soap:Envelope>
```

Atributul *actor*

Un mesaj SOAP paote fi rutat prin mai mulţi intermediari. Se foloseşte un atribut *actor* care specifică adresa intermediarului. Acest intermediar trebuie să elimine informaţia de intermediar care îl priveşte, dar poate să adauge la rândul lui altă informaţie de acest tip.

13.4.2 Elementul *body*

Un mesaj SOAP poate avea un singur element de tip *body* al cărui conţinut poate să fie structurat XML sau simple şiruri de caractere, orice care nu invalidează conţinutul documentului XML conţinător. Pe baza acestui conţinut, un mesaj SOAP poate fi privit ca un mesaj orientat pe procedură sau ca unul de tip document.

Pentru primul caz, să presupunem că SW cere apelarea unei metode de adunare a 2 numere:

```
public int Add(int x, int y)
{
  return x + y;
}
```

Mesajul SOAP la apelarea acestei metode este:

```
<?xml version="1.0"?>
<soap:Envelope
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <Add>
      <x>1</x>
      <y>2</y>
    </Add>
  </soap:Body>
</soap:Envelope>
```

Răspunsul generat de serviciul Web este:

```
<?xml version="1.0"?>
<soap:Envelope
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <AddResult>
      <result>3</result>
```

```
    </AddResult>
  </soap:Body>
</soap:Envelope>
```

Există suport pentru transmiterea tipurilor de date definite de utilizator, a array-urilor, definirea parametrilor ca fiind de tip valoare/referinţă/ieşire, a grafurilor de obiecte, etc.

Bibliografie

[1] *C# .NET. Web Developer's Guide*, Turtschi Adrian, DotThatCom.com, Werry Jasson, Hack Greg, Albahari Joseph, Nandu Saurabh, Lee Wei Meng; Syngress Publishing, 2002

[2] *Programming C#*, Jesse Liberty; O'Reilly, 2001

[3] *.NET Framework Essentials*, Thia Thuan, Lam Hoang Q.; O'Reilly, 2001

[4] *C# Language Specification*, ECMA TC39/TG2, Octombrie 2002

[5] *A Programmer's Introduction to C#*, Eric Gunnerson, APress, 2001

[6] *A Programmers's Introduction to C#*, Syngress Publishing, 2002

[7] *Professional ADO.NET Programming*, Wrox, 2001